Università degli Studi di Padova

Dipartimento di Ingegneria dell'Informazione

Master's Degree in
ICT for Internet and Multimedia

Using Autoencoders in Homogeneous and Heterogeneous Transfer Learning Configuration Auto-Tuning Problems

Relatore:
Prof. Nicola Ferro

Laureando:
Alessandro Buratto
2019041

Correlatore:
Dott. Stefano Cereda

Anno Accademico 2021/2022
Data di Laurea 13 ottobre 2022

Ai miei nonni

Abstract

Performance auto-tuning is a trending topic in ICT applications. Historically the optimization of complex systems has been performed by an human actor, but with the growing complexity of current day information systems and the need of fast and effective optimization there is a strong need to automate this process. We propose a performance auto-tuning algorithm based on Collaborative Filtering and Auto Encoders in order to transfer useful knowledge between an already tuned source application and a novel target one. We test our method with datasets composed of data points collected with open-source benchmark suites and DBMS commonly employed in enterprise settings. We find that our algorithm is both fast and effective in finding new configurations to apply in the system to be tuned.

Keywords: Autoencoder, Transfer learning, Auto-tuning

Acknowledgements

I would like to start by thanking Professor Nicola Ferro for supervising my thesis work and giving interesting and useful suggestions in the development of this project.

I would like to thank all the people working at Akamas for allowing me to develop my internship with them and especially Stefano Cereda who has been my Supervisor throughout these last seven months giving me excellent suggestions, corrections and support in developing this work.

Finally I would like to thank my family and friends for their presence and encouragement throughout these years.

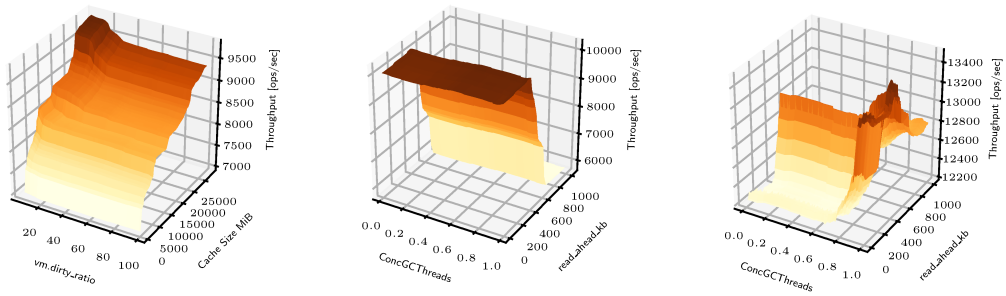
Contents

Chapter 1

Introduction

Optimization is a trending problem in all the ICT applications from networks routing to complex software stacks and distributed resources allocation. Complex systems, most of the times, behave as black boxes so it is difficult to find correlations between settings adopted in an optimization session and the resulting performance obtained. Moreover an IT system is often subject to a varying amount of load (also known as workload) which considerably alters the impact of tunable parameters on the performance. This phenomenon can be seen in Figure 1.1 where different DBMS systems experience varying workload conditions and it is clear that the same parameter has really different effect on the throughput (the performance metric considered, represented in the images as a 3D mesh) which varies considerably. Historically, optimization of complex was a task reserved to human actors, who with their experience were able to optimize correctly the system considered.

In this work we will focus on automatic performance optimization, which aims at optimizing some Key Performance Indicators (KPI) which can be as diverse as throughput, latency, response time, number of simultaneous calls to the service and many others. In automatic performance optimization there are many methods that have been used in the literature. One of the most popular and efficient is Bayesian Optimization (BO) [38]. Other commonly used techniques are multi armed bandits [31] and methods derived from the field of Recommender Systems [12]. Since the optimization of new architectures is generally very expensive both in time and economic terms, there is a great interest in Transfer Learning techniques in order to reuse some previously acquired knowledge of a system to allow a faster tuning for new applications by leveraging the similarities between their domains. Transfer Learning divides itself into two main categories according



(a) MongoDB with update-heavy workload **(b)** Cassandra with update-heavy workload **(c)** Cassandra with read-only workload

Figure 1.1: Throughput of a DBMS as a function of some of its tunable parameters. In (a) we use MongoDB and vary its cache size and the `vm.dirty_ratio` Linux parameter. In (b) we use Cassandra, varying the number of threads for the concurrent garbage collector of the JVM (expressed as a percentage of the available cores) and the `read_ahead_kb` parameter of the Linux kernel. In (c) we repeat the experiment of (b), but using a different YCSB workload.

to the domain structure and distribution of source and target datasets: Homogeneous Transfer Learning "only" faces the problem on how to align different distributions in order to make a useful transfer of knowledge between domains; Heterogeneous Transfer Learning focuses also its attention in having a way to compare effectively two domains that have a really different domain structure between themselves. This is generally done by a mapping in a different space from the original one, but it is not always the adopted strategy [42, 44]. Heterogeneous Transfer Learning is subdivided into two main approaches, namely Asymmetrical Transfer Learning and Symmetrical Transfer Learning. In our work we will use Symmetrical Transfer Learning to map both a source and target domain in a same space where it is easier to apply known methods to find and leverage similarities between points. In this work we focus on a symmetrical approach to compare configurations between the source and target domain. In order to do this, we use denoising Auto Encoder (AE)s and denoising Variational Auto Encoder (VAE)s trained in a fully unsupervised way. We also explore the possibility of training the target model during the optimization process considering that it is very likely that "*in the wild*" a tuner could not have many data points for the target system. To efficiently select points in the common domain between source and target applications we take an optimization algorithm built using Collaborative Filtering (CF) technique from the Recommender Systems field.

This thesis work is divided as follows: In Chapter 2 we present in detail the background needed to understand our work and present the relevant state of the art techniques that have been used in the literature. In Chapter 3 we describe our

method describing step by step the choices we made in designing the algorithm. In Chapter 4 we discuss the metrics we use to evaluate the performance of the proposed approach and we will give a high level presentation of the datasets we adopted for our experiments. In Chapter 5 we present and discuss the results we obtained in our experiments and we give some considerations on the tuning time our method needs in order to give new configurations to try. In Chapter 6 we draw the final conclusions of our work and propose some future extensions to this thesis.

Chapter 2

Background and State of the Art

In this chapter we are going to lay the background knowledge necessary to understand our work and report to the best of our knowledge the current state of the art for performance autotuning. Section 2.1 focuses on performance optimization, motivates the need for an automatic approach to optimization and presents some State of the Art algorithms that solve this issue. Section 2.2 explains what is transfer learning and why it is important in modern day Machine Learning applications. Section 2.3 explains what are recommender systems and their main algorithms. Section 2.4 presents the DNN models we are going to use in our work. Finally, Section 2.5 presents BO and gives some examples of application in the literature.

2.1 Performance Optimization

Performance optimization is a trending topic both in the research and commercial field. The main goal is to optimize a set of KPI, which are measurable attribute of the target system as execution time, response time, throughput, energy consumption, jitter and many others. In this work Key Performance Indicators (KPI) are named performance metrics or just metrics. These are what we aim to optimize by changing the configuration parameters of a system. With the word system we refer to any application with all the IT stack that sits underneath what the final user will experience. Each system has many configuration parameters that can be seen as little tunable knobs that change single aspects of this complex model. For instance let us imagine we need to minimize the response time of a database built in Java. The system can be seen as the Java Virtual Machine (JVM) that runs the program, the Operating System (OS) of the machine where the code

is running, the container environment, etc. Each of these compartments have their own set of parameters that can be modified in order to affect the final performance metric of interest. In most cases there is not a clear correlation between the configuration parameters and the metrics. Historically, the process of performance optimization was delegated to a very specialized human who took advantage of his or her knowledge of the system architecture to produce new efficient configurations. This performance optimization loop is very time consuming and it is bound to the level of expertise of the agent, thus it is important to find smart strategies in order to get good sets of parameters with a small investment in optimization time. To achieve this goal there is a branch of performance optimization, which is performance auto-tuning. In performance auto-tuning the human agent is taken out of the optimization loop, in its place there are clever algorithms designed to be both fast and data-efficient. There are many possible techniques to achieve this goal. Taking for example compilation flags optimization Chen et al.[15] discuss about the use of Iterative Compilation (IC), which consists of trying many sets of configuration parameters until a good enough set is found. Approaches as OpenTuner [4] try many different approaches, including hill climb algorithms and multi-armed bandits, and then focus the resources on optimization on the better performing methods. Other works state that a simple random search in the tunable domain is sufficient to find good configurations in a relatively short time [11, 1, 15]. There is a big family of methods which try to take advantage of knowledge made on previous tuned programs to improve speed of convergence on new programs to be tuned. Static code features have been used to guide the optimization process in [1]. Anwar et al. have tried to use association rules to find similarities between software datasets [5], this knowledge can then be leveraged to improve performance autotuning between these sets of applications.

In this work we build on top of extensions of IC techniques, namely Collaborative Filtering (CF) tuner presented in 2.3.2.

2.2 Transfer Learning

Transfer Learning is an important field of Machine Learning. It studies how to obtain a high-performing model for a target domain starting from knowledge already obtained from a source domain. This is to counteract the scarcity of data points for the target domain taking advantage of an extensively labeled

source domain. Another way to look at the data domains in a transfer learning environment is to think that the source and target data live in two separate sub-domains linked by a high-level common domain [42]. A simple example in a human relatable context is that of two people wanting to learn how to play guitar: the first one already knows how to play the piano, the other one has never touched a musical instrument. Intuitively, the one who already knows how to play the piano will need less effort in learning the guitar than the other one simply because he or she already has some knowledge of music. In this example we can see *playing the piano* and *playing the guitar* as two sub-domains of *musical instruments*. Formally, for a domain $\mathcal{D}$ defined by a feature space $\mathcal{X}$ and a marginal probability distribution $p(X)$, where $X = \{x_1, \ldots, x_n\} \in \mathcal{X}$, a task $\mathcal{T}$ is defined by a label space $\mathcal{Y}$ and a predictive function $f(\cdot)$ that is learnt from the feature vector-label pairs $\{x_i, y_i\}$ where $x_i \in \mathcal{X}$ and $y_i \in \mathcal{Y}$. We now define the source domain as $\mathcal{D}_\mathcal{S} = \{(x_{S1}, y_{S1}), \ldots, (x_{Sn}, y_{Sn})\}$ where $x_{Si} \in \mathcal{X}_\mathcal{S}$ is the i-th data instance of $\mathcal{D}_\mathcal{S}$ and $y_{Si} \in \mathcal{Y}_\mathcal{S}$ is its corresponding label, further more we define $\mathcal{T}_\mathcal{S}$ as the source task and $f_S(\cdot)$ as the source predictive function. Symmetrically we define $\mathcal{D}_\mathcal{T}$, $\mathcal{T}_\mathcal{T}$ and $f_T(\cdot)$ for the target domain. Finally transfer learning is defined as improving the target predictive function $f_T(\cdot)$ by using the related information from $\mathcal{D}_\mathcal{S}$ and $\mathcal{T}_\mathcal{S}$, where $\mathcal{D}_\mathcal{S} \neq \mathcal{D}_\mathcal{T}$ or $\mathcal{T}_\mathcal{S} \neq \mathcal{T}_\mathcal{T}$. With this definition we can subdivide transfer learning into two main big categories:

- **Homogeneous Transfer Learning**: this happens when $\mathcal{D}_\mathcal{S} = \mathcal{D}_\mathcal{T}$, to be more specific when $\mathcal{X}_\mathcal{S} = \mathcal{X}_\mathcal{T}$ and the marginal probability distributions of source and target are different. In a practical example this happens when we try to train a classifier for images of dog breeds using an already trained classifier that has the same input structure (image size, max-min values, etc.).

- **Heterogeneous Transfer Learning**: happens when $\mathcal{D}_\mathcal{S} \neq \mathcal{D}_\mathcal{T}$, or more simply when $\mathcal{X}_\mathcal{S} \neq \mathcal{X}_\mathcal{T}$. This means that the source and target domains are formed with different feature domains.

The latter is generally the case for software and IT architecture optimization since not all the considered applications run on the same OS, have been written in the same language, compiled with the same compiler and other technological similarities. For this reason we are going to focus our attention on heterogeneous transfer learning techniques as they better suit our needs and can eventually still be used for the homogeneous case as we will later see in Section 5.3.

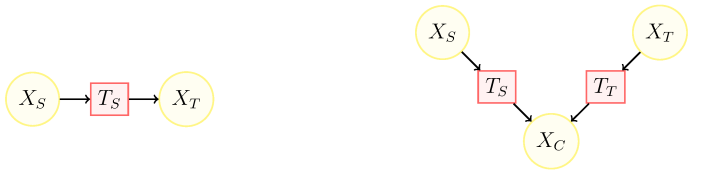


Figure 2.1: Asymmetric Feature Transformation and Symmetric Feature Transformation flow diagrams. X_S, X_T and X_C are the source, target and common domain feature spaces respectively. T_S and T_T are the source and target transformations

2.2.1 Heterogeneous Transfer Learning

The majority of the methods employed in heterogeneous transfer learning focus on aligning the input space of the source and the target domain into a common space with the assumption that the source and target domain distributions $p(X_S)$ and $p(X_T)$ are the same. If this condition is not verified, then some domain adaptation steps are needed to align the two probability distributions. There are four main transfer categories that have been identified in Transfer Learning [42, 34]. Speaking of the Transfer Category of feature-based learning there are two main approaches that can be employed:

- **Asymmetric Feature Transformation**: this type of Transfer Learning is applied to re-weight the source domain to resemble the target domain. It is called asymmetric because the transformation is applied only to the source data.

- **Symmetric Feature Transformation**: with this approach a different transformation is applied to both the source and target data to discover a common latent feature space while simultaneously reducing the marginal distribution between the domains.

Both the feature transformations workflows can be visualized in Figure 2.1 where the ingredients to make these kind of transformations are clearly visible. In our setup we will be using the Symmetric Feature Transformation because we aim at obtaining a common feature space between the source and target where finding similarities between the two might be easier.

In literature there have been many successful attempts in transferring knowledge between different domains in many fields such as image recognition, multi-language classification, human activity classification, software defect classification and many others. Shi et al. [39] proposed a spectral method to perform effectively transfer learning in case the source and target domain have a different

space ($\mathcal{X}_\mathcal{S} \neq \mathcal{X}_\mathcal{T}$), the marginal distributions $p(X_S)$ and $p(X_T)$ are different and also the output space between source and target is different ($\mathcal{Y}_\mathcal{S} \neq \mathcal{Y}_\mathcal{T}$). The approach uses a spectral mapping technique to align source and target domains, then a clustering based sample selection method is applied to reduce the marginal distribution differences and finally a Bayesian method is used to resolve the differences in the output space. In the experiments this method outperforms the baseline method used, but there is not a detailed description of the baseline used. Duan [18] proposed a method to perform domain adaptation using a single source domain with labeled samples and a less populated target domain with only a few labeled samples. The approach finds a matrix H that combines a transformation matrix P for the source domain where the target features are zeroed out and a transformation matrix Q which contains only the common features and zeroes the source and target ones. Tests using this approach on the task of image classification and text classification show that this method can achieve good result. Li et al. [33] extends the work of Duan [18] and proposes a decomposition of the H matrix into rank-one positive semi-definite matrices that allow for Multiple Kernel Learning solvers to be used. In the training process, pseudo labels for the target data are estimated and used in learning the final target classifier. This extension improves over Duan and achieves much better results than Shi [42]. Other approaches target the transfer learning problem in the homogeneous case using Bayesian Optimization (BO) with a combination of multiple kernels [28, 41, 40]. These methods generally achieve good results when a small amount of kernels are used (no more than 3). It is also important to note that these methods, with an accurate choice of kernels can be applied to the heterogeneous case given that they are aided by some other technique to deal with different input spaces.

2.3 Recommender Systems

Recommender Systems are a sub-category of information filtering systems[1] that provide suggestions for items that are more likely to be relevant to a specific user [35, 36, 12]. They become very useful when an individual needs to select an item from a very large possible set of options a service may offer. There are two main approaches to implement Recommender Systems:

[1] automatic systems that remove unwanted information from an information stream before the presentation to the user.

- **Collaborative Filtering**: the main idea behind this approach is that users that made similar choices in the past will continue to make similar choices in the future.

- **Content-based Filtering**: this approach need a description of the item and an user's preference profile in order to make recommendations.

In literature both these approaches have been successfully used to implement successful performance auto-tuning solutions: Al Baity et al.[3] managed to implement a Content-based Filtering approach that obtained tuning configurations that reduced execution times of compiled programs relative to simple use of the `-O3` compilation flag; Christoforidis et al. [16] implemented a Collaborative Filtering technique to obtain configurations for Intel CPUs that greatly reduced the energy consumption in just a few iterations.

Our work takes its foundations on the paper by Cereda et al. [12] that proposed a Collaborative Filtering approach to find configurations of compiler flags that sped up execution times of the compiled programs.

2.3.1 Top Popular (TP)

Top Popular (TP) is the simplest Recommender System algorithm which simply suggests the most popular items [17]. The assumption that configurations that are beneficial to most programs are also beneficial to a new program is generally reasonable. The relevance of a configuration applied to a program p is calculated as:

$$r_p(\boldsymbol{x}) = \frac{f_p(\boldsymbol{x})}{f_p(\boldsymbol{x_0})} - 1$$

Where $f_p(\cdot)$ is the function that gives the value for the considered KPI and $\boldsymbol{x}_0$ is the configuration we keep as baseline. The relevance score of a configuration for a new program given a knowledge base of previously tuned programs $Q = \{q_0, q_1, \dots\}$ is calculated with the following equation [12]:

$$\tilde{r}_p(\boldsymbol{x}) = \tilde{r}(\boldsymbol{x}) = \frac{\sum_{q \in Q} r_q(\boldsymbol{x})}{|Q|}$$

Where p is the new program we consider and $\boldsymbol{x}$ is the configuration vector we apply. It is important that the program we need to tune does not affect the calculation for the relevance, this means that the relevance is a property of the configuration itself in the knowledge base Q.

After calculating the relevances the algorithm proposes new configurations in the order given by sorting the relevances scores.

2.3.2 Collaborative Filtering (CF)

The algorithm proposed in [12] is a modification of Collaborative Filtering that uses also RM, a reaction-based characterisation methodology. To apply this method we expect that two programs that benefit from the same set of configurations will have similar behaviour when other sets are applied.

Since this algorithm aims at measuring the similarity of the programs we need to update the normalized relevance as:

$$\tilde{r}_p(\boldsymbol{x}) = \frac{\sum_{q \in NN_{kp}} s_{pq} r_q(\boldsymbol{x})}{\sum_{q \in NN_{kp}} s_{pq}}$$

Where NN_{kp} is the set of k programs most similar to p (thus they can be viewed as the k Nearest Neighbours) and s_{pq} is a similarity measure between programs.

The algorithm works as following: given the knowledge base $Q = \{q_0, q_1, \dots\}$ and the program p to be tuned as in Section 2.3.1, at the first iteration we try to evaluate p with the baseline $\boldsymbol{x}_0$. As a second evaluation we try the first configuration found by the TP algorithm. We now apply RM, after bringing all the baselines to 0 we define the similarity of two programs as the distance of the relevance scores they received on the same configurations. Defining the sequence $\{\boldsymbol{x}_i\}_{i=1}^n$ as the sequence of configurations explored until iteration n we compute the euclidean distance as:

$$d_{pq} = \sqrt{\frac{\sum_{i=1}^{n} \left(r_p(\boldsymbol{x}_i) - r_q(\boldsymbol{x}_i)\right)^2}{n}}$$

At this point we compute the similarity $s_{pq} = \frac{1}{d_{pq}}$ and we use it to calculate the relevances with the updated formula. With these new relevances we find the next configuration to apply and after evaluating it we need to recompute the distances, the relevances, suggest the next configuration and so on.

This algorithm is much more complex than TP as it updates the distances according to already evaluated configurations. A big advantage of this approach is that it is workload-dependent and thus can easily adapt to changing load on the application. For this reason we build our approach on top of this algorithm to try to extend the approach to domains which have really different configuration spaces from each other.

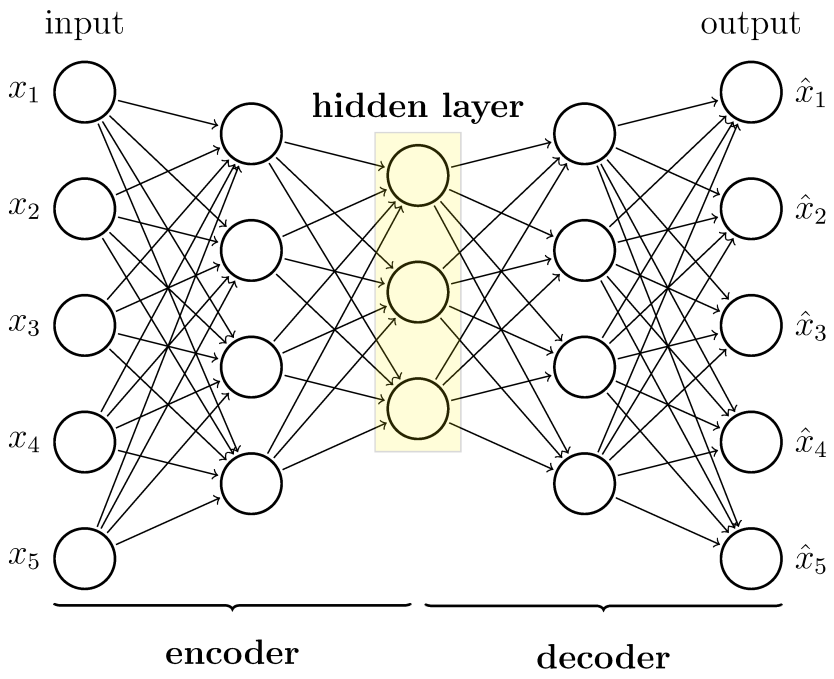


Figure 2.2: Architecture of an Auto Encoder (AE)

2.4 Deep Neural Network Models

We hereby present some of the useful architectures for deep learning that will be used in our work. We aim at encoding and reconstructing parameter configurations following the Asymmentric Feature Transformation paradigm presented in Section 2.2.1. To do this we decided to use Auto Encoder (AE) as our Deep Neural Network (DNN) models. These architectures fall under the label of unsupervised learning, this means that they do not need labeled data in order to be trained as they can learn *on their own* without the need of having a *teacher* that tells them if they are doing things correctly or not. These architectures are composed by fully connected layers[2] with some kind of activation function in between (Sigmoid, Hyperbolic Tangent, ReLU, ...). The following subsections explain in detail how these architectures are composed and how they are trained.

2.4.1 Auto Encoder (AE)

The Auto Encoder (AE) is a deep learning architecture for unsupervised learning characterized by fully connected layers divided into two main structures: an encoder that is responsible for an encoding operation into the hidden layer and a decoder trained to reconstruct the original input (Figure 2.2). It is important to

[2]each node from a layer is connected to all other nodes in the following one.

note that the hidden layer is in common between the encoder and the decoder. The encoded sample stored in the hidden layer is often referred to as the hidden representation of the input [23]. Mathematically, sample $\boldsymbol{x} \in \mathbb{R}^n$ is encoded into sample $\boldsymbol{z} \in \mathbb{R}^m$ with for some n, m (n is the dimensionality of the input space and m is the size of the encoded latent space) by a Multi Layer Perceptron (MLP) that performs a non linear operation E parametrized by the set of learnable parameters ϕ: $\boldsymbol{z} = E_\phi(\boldsymbol{x})$. Similarly $\boldsymbol{z}$ is transformed back into the original space $\mathbb{R}^m$ by another MLP that performs another non linear operation D parametrized by the set of learnable parameters θ: $\boldsymbol{x'} = D_\theta(\boldsymbol{z})$. We train both the MLP at the same time to minimize the L2 loss between sample $\boldsymbol{x}$ and the reconstructed sample $\boldsymbol{x}'$.

$$\min_{\theta,\phi} L(\theta,\phi), \text{where } L(\theta,\phi) = \frac{1}{N}\sum_{i=1}^{N} ||x_i - D_\theta(E_\phi(x_i))||_2^2$$

2.4.2 Denoising Auto Encoder (dAE)

The deep denoising auto encoder is not a new concept, it has been used in the literature to improve the reconstruction quality of deep kernelized auto encoders [27] by introducing an user definable amount of noise in the input. The difference from a standard AE is only in the training process. The noise, typically AWGN, is added calculated and added to each batch independently at every epoch. In order to perform efficiently data augmentation the noise targets random individual values in the training points creating many different noisy input sequences that help in avoiding overfitting of the model. It has to be noted that the model tries to reconstruct the original input and not the noisy one. This differs from the traditional data augmentation technique where the model is trained to reconstruct correctly the noisy input. Mathematically, given $\boldsymbol{x}$ as the original sample, $\hat{x}$ the sample with added noise, $\boldsymbol{x'}$ the reconstructed sample and the same symbols of 2.4.1, the loss to minimize during training is

$$L(\theta,\phi) = \frac{1}{N}\sum_{i=1}^{N} ||x_i - D_\theta(E_\phi(\hat{x_i}))||_2^2$$

This technique of data augmentation helps the convergence of the training process in case of small datasets and has the increased benefit of obtaining a trained AE which is considerably more robust to noisy inputs in the inference phase.

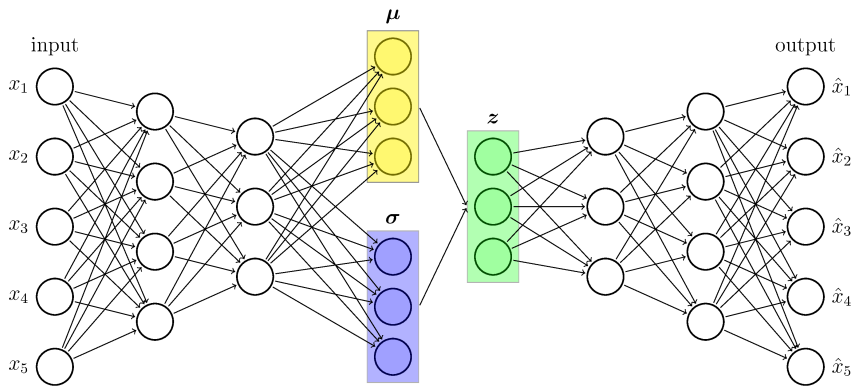


Figure 2.3: Architecture of a Variational Auto Encoder (VAE)

2.4.3 Variational Auto Encoder (VAE)

The Variational Auto Encoder (VAE) is a variation over the vanilla AE designed to overcome the poor quality of the results obtained by decoding configurations obtained by sampling in the innermost layer [30, 29]. The main theoretical difference between this model and the AE lies into the fact that now the model does not learn a latent representation of the input, but rather learns a probability distribution of the input [23]. This architecture, shown in Figure 2.3, splits the hidden layer into two set of nodes: one symbolizes the mean of a multivariate gaussian distribution, the other the variance. To generate a sample to be decoded we sample from the multivariate gaussian distribution parametrized by the two hidden layers. In order to still be able to train the model by backpropagation of the gradient, we need to remove the non linearity introduced by sampling from the gaussian distribution. To achieve this we take advantage from the reparametrization trick. The main idea is to treat sampling as noise, to do this we add a normalized gaussian noise vector. Mathematically, given $\boldsymbol{\mu}$ as the means vector and $\boldsymbol{\sigma}$ as the variance vector, $\epsilon \sim N(0, 1)$ is the stochastic noise vector, the reparametrized sample $\boldsymbol{z}$ can be written as

$$\boldsymbol{z} = \boldsymbol{\mu} + \boldsymbol{\sigma} \cdot \boldsymbol{\epsilon}$$

Figure 2.4 shows graphically how this technique allows the use of backpropagation since it removes the stochasticity from node $\boldsymbol{z}$ allowing the calculation of the gradient.

To effectively train this kind of model the simple L2 loss described for the AE in 2.4.1 is no longer sufficient. For this reason we need to add another term to the loss function called the Kullback Leibler divergence [32] D_{KL}. This is a

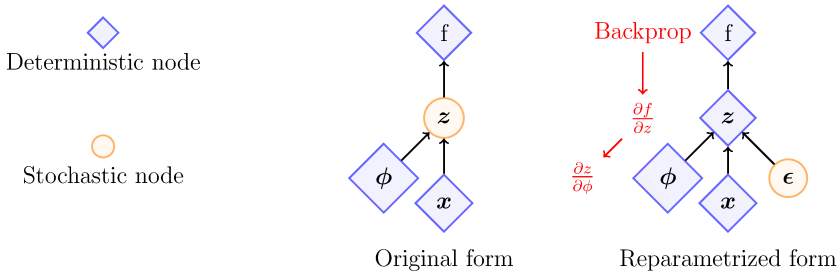


Figure 2.4: Diagram of the reparametrization trick

statistical distance which measures how much two distributions are different from each other. With this new distance the loss function becomes:

$$loss = ||\boldsymbol{x} - \boldsymbol{x'}||^2 + D_{KL}(N(\boldsymbol{\mu}, \boldsymbol{\sigma}), N(0, I))$$

Where $\boldsymbol{x}$ is the original sample

For our experiments, we implemented a more advanced version of this model called β-VAE [24] which adds a coefficient $\beta >= 1$ in front of the KL loss in order to obtain increased quality reconstructions by forcing more disentangled representations. Notice that if $\beta = 1$ it is the same model as the vanilla VAE

$$loss_\beta = ||\boldsymbol{x} - \boldsymbol{x'}||^2 + \beta D_{KL}(N(\boldsymbol{\mu}, \boldsymbol{\sigma}), N(0, I))$$

2.4.4 Denoising Variational Auto Encoder (dVAE)

Denoising Variational Auto Encoder (dVAE) builds on top of the same idea for data augmentation presented in Section 2.4.2, improving the reconstruction accuracy when dealing with noisy inputs. It has been demonstrated that training the VAE with this procedure achieves a theoretical tighter bound to the theoretically achievable loss of a vanilla VAE [25].

2.5 Bayesian Optimization

In our work we leverage BO by tuning our DNN hyperparameters with Optuna [2]. To give a better understanding of how this tool works we now present Bayesian Optimization.

Bayesian Optimization (BO) is a powerful mathematical method designed to solve complex optimization problems involving the global maximization or minimization of a black box function [38]. This kind of functions do not have a

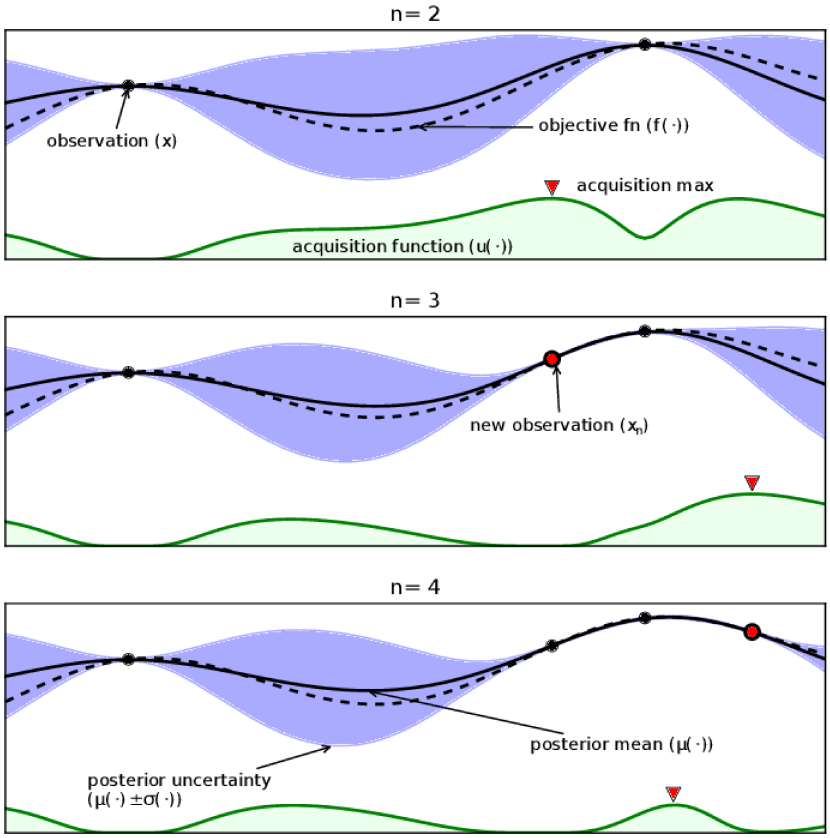


Figure 2.5: Graphical representation of BO optimization procedure showing three consecutive iterations. The plots show the mean and confidence intervals estimated with the stochastic model. The acquisition function is also represented in green, it has high values when the model predicts a high objective and where it has high uncertainty. Credits to [38]

simple closed form expression, but can be evaluated at any point in their domain $\mathcal{X}$. The evaluation of a black box function at a point $\boldsymbol{x} \in \mathcal{X}$ produces stochastic outputs $y \in \mathbb{R}$ such that $\mathrm{E}[y|f(\boldsymbol{x})] = f(\boldsymbol{x})$. BO requires two main ingredients to function properly:

- a probabilistic surrogate model, which consists of a prior distribution that captures the beliefs on the behaviour of the objective function, uses the data obtained during the exploration and gives the posterior distribution;

- an acquisition function that leverage the uncertainty of the posterior in order to guide the exploration;

Another important ingredient in BO is a loss function introduced to describe how optimal is a sequence of queries.

Figure 2.5 shows graphically how a few iterations of BO are carried out. It is important to note that the acquisition function does not have its maximum when

the model has big uncertainty, but it moves the maximum towards area with high uncertainty and where the model predicts a high objective value.

Most of the time the preferred probabilistic model is a Gaussian Process (GP). In statistics a GP is a stochastic process[3] for which every variable is a multivariate normal distribution, in other words any linear combination of them is normally distributed. A peculiarity of these stochastic processes is that they can be fully described by their second order statistics, *i.e.* their covariance matrices also known as kernels [26]. A very common choice for a kernel is the Matérn Kernel as used in many publications on BO [13, 31].

BO has seen many applications in the literature: it has been used for A/B testing when there is a strict query budget since it is very efficient; it has been used in recommender systems of news articles [14]; many applications involve tuning hyperparameters of Machine Learning models [7, 20], and BO has also been used to find the best algorithm to apply to solve a given task.

[3] a collection of random variables.

Chapter 3

Proposed Model

In this Chapter we are going to describe how we initialized and trained our DNN models, then we present our modified version of the Collaborative Filtering (CF) algorithm shown in Section 2.3.2 in which we will explain how our method deals with source and target domains having different sets of tunable parameters. The full pseudocode of our approaches is in Algorithm 1 for AeTuner and Algorithm 2 for AeNTTuner

3.1 Initialization Algorithm

The first step to be performed in our method is to select and train the source and target model. Due to the fact that our aim is to project the source and target domain into a common space, we opt for using an Auto Encoder and a Variational Auto Encoder as we have anticipated in Section 2.4. In order to build these models we need to define their hyperparameters[1] in a way that allows the model to learn how to reproduce samples from the source and target domain in the most accurate way possible. It is in fact well known that a DNN model with a random set of hyperparameters will not achieve results as good as a model built with a set of hyperparameters that favor the learning of a given dataset [23]. For this reason we choose to use Optuna [2] to find a good hyperparameter configuration for our model. To do this, we have to train our candidate models for a small number of epochs with the source and target dataset. The goal metric to be minimized in our case is the validation loss calculated depending on the

[1] *i.e.* everything that is not the learnable parameters (the weights) of the connections in the DNN, for instance the number of hidden layers, their sizes, the type of optimizer, the learning rate of the optimizer, the batch size if the training is done with mini-batch technique, ...

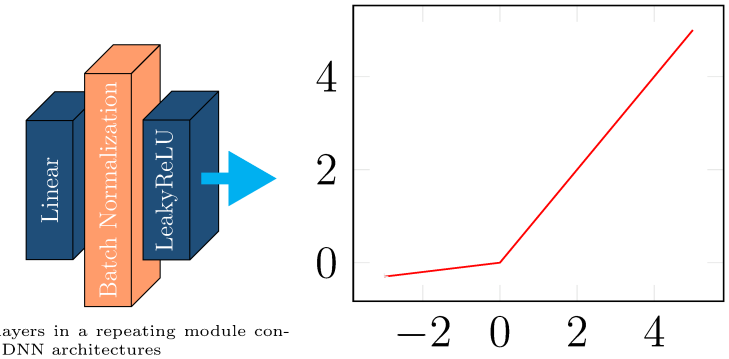
considered model according to the equations reported in previous sections. With this setup, Optuna performs BO for us under the hood and our only concern is to choose which set of hyperparameters to optimize. In order to speed up this repeated training of DNN models, we attach a median pruner to the optimizer that stops automatically the training process if the validation loss at a given epoch is higher than the median loss achieved by other models at the same iteration. It has to be noted that we did not tune all the same hyperparameters for our source and target model: we avoided tuning a second time the innermost layer dimension (latent dimension) for both the source and target model. As our goal is to obtain a common space between source and target, we simply kept the same value obtained optimizing the target, as the target's decoder will be the only one used in the tuning process.

We also need to adapt the input data to our model: we want to avoid having largely different values at nodes inputs (*e.g.* one node has 1000 at input and another one -0.23) as this would make the training process much more tedious. For this reason we normalized all the points in the dataset using Skopt[2] Spaces. This implementation allows for an easy normalization and un-normalization of our parameter spaces. An important implementation detail is that the One-Hot encoding performed with categorical type of inputs does not follow the common rule of allocating two different binary values when there are only 2 options, but instead it uses binary code to perform the normalization.

The simplified unit of our model is a sequence of a linear layer with LeakyReLU activation function, as it will help in reducing vanishing gradients in case we need a really deep model to correctly represent our datasets, and a Batch normalization layer. This type of layer acts to reduce the standard deviation of weights when tuning with mini-batches. It is a really efficient regularization technique and, generally speaking, it has the side effect of reducing the capacity of the model to fully learn the dataset. In our experiments we noticed the opposite effect, possibly due to the prior normalization we applied to the points, for this reason we kept it in the final model.

After this first model selection step, we take the best set of hyperparameters found by Optuna for the source and target and we train the source and target models to fit their respective dataset. In order to obtain better quality results in the reconstruction of configuration patterns we train a DNN model for every application in each dataset. During the real training process we let the model

[2]https://scikit-optimize.github.io/stable/



(a) Sequence of layers in a repeating module constituting our DNN architectures

(b) LaekyReLU activation function plot

Figure 3.1: The sequence of layers composing the module which is repeated in the architecture of our DNN models (a) and a plot of the LeakyReLU activation function which is adopted to avoid gradient vanishing in deeper models (b)

train for an arbitrary amount of epochs until the early stopping criterion is met. This allows us to obtain correctly trained models without having to worry about an overfitting model.

In Algorithm 2, we don't have full access to the test dataset. In this case we choose the same set of hyperparameters of a previously trained model for that dataset if we already had an optimization session on it, otherwise we choose hyperparameters similar to the ones of the source domain paying attention at keeping the learning rate and the batch size the same since most of the time they are the more important hyperparameters we can tune. At the same time we need to initialize the weights of the target DNN in a way that minimizes the amount of errors performed in the first iterations of our algorithm. We find that a good initialization is the uniform inizialization proposed in [22] where the weights assume values according to the following equation:

$$w = gain \times \sqrt{\frac{6}{fan\_in + fan\_out}}$$

where *gain* is calculated accordingly to the activation function used by that node. In our work we used `LeakyReLU` and for this reason we need to use

$$gain = \sqrt{\frac{2}{1 + negative\_slope^2}}$$

Fan_in and fan_out refer to the number of edges each node has in input and in output.

After this initial training of the source and target model we need to fill the Knowledge Base (KB) with all the points in the source domain. To improve the

efficiency we store the source points in their encoded form. For this reason we have the source model fully pretrained. The filling of the knowledge base proceeds as follows: for every source dataset, for every benchmark in this dataset and for every workload in each benchmark we evaluate all the configurations present in the workload, then we normalize them in the source domain[3] and finally we encode them using the encoder portion of the model into the common space. In this form we finally include them in the KB. Formally given a set of points in the source domain $\boldsymbol{x} \in \mathbb{R}^n$ were n is the dimensionality of the source domain, we normalize them obtaining points $\hat{\boldsymbol{x}}, \hat{x}_i \in [0, 1]$ and then we use the trained encoder of the source DNN E_S to map the normalized points in the common domain between source and target $\mathbb{R}^c$ with c being the dimensionality of this space and generally it holds that $c \neq n$.

3.2 Tuning Algorithm

3.2.1 AeTuner

With all the initialization steps described in Section 3.1 we can now proceed into the tuning of the target suite. Our approach is based onto the CF tuning algorithm [12] we have mentioned in Section 2.3.2. The first step is to take a predefined number of points from the KB as we will use them for our initial points in the Unified Relationship Matrix (URM). Formally given a knowledge base KB we take a number of points n_p where each point $\tilde{\boldsymbol{x}} \in X_S \subset \mathbb{R}^c$ where c is the dimensionality of the common space between source and target. After this, we can start working with the target space. We begin by evaluating the initial points X_0 given to the algorithm in the original target domain. After the evaluation we need to encode them in the common domain. In order to do this we normalize the configuration similarly as we did in Section 3.1 and then we apply the encoder of the target model E_T, the result is then added to the KB and the original point is added to the list of already evaluated points $eval\_xs$. We now enter the main optimization loop that will continue until we reach the number of iteration we have been given at initialization. The first action we need to perform is to move the points X_S in the normalized target domain. To do this we use the decoder part D_T of the target autoencoder. Mathematically the operation is as follows $\hat{\boldsymbol{x}} = D_T(\tilde{\boldsymbol{x}}), \forall \tilde{\boldsymbol{x}} \in X_S$, where D_T is the target model's decoder and $\hat{x}$ is the point

[3]That is because our DNN models are trained with only normalized inputs for each of the input nodes.

in the normalized target domain. After this we move on by applying CF using the same sequence of actions we would do if we were doing the simple CF algorithm, but using our newly defined set of points. We compute the weights to be assigned to the points in the decoded X_S (which are the similarities mentioned in Section 2.3.2), we compute the ranking of all points and we order them according to this result. Having now the ranking of the points, we need to decode them in normalized target domain using D_T and we unnormalize them obtaining valid configurations we can apply to the system. At this point we search for points we have not explored before by iterating through the list of all ranked points. When we find a new point we evaluate it, encode it once more using E_T and we add it to the KB.

In Section 5.3 we will also use a slightly different version of this algorithm to obtain a theoretical maximum achievable by removing the decoding error introduced by the autoencoder models. The only difference is that we do not use the decoder D_T at any point. In its place we need to keep a separate KB containing the same points of the one in common domain and we use it to get the original configurations as they are at the same indexes.

3.2.2 AeNTTuner

AeNTTuner follows a similar tuning procedure as described in the previous Subsection. The main difference is that we do not have a completely trained target model from the start. The training happens as the tuning procedure advances. We arbitrarily decide that we perform a training cycle for the target model once every time we reach a number of points in the set of already evaluated points that allows us to subdivide our data in an integer number of training and validation batches. This leads to a faster initialization phase, but then some iterations in the tuning phase appear to be much slower due to this extra step. As we did for AeTuner, we performed a similar modification to the algorithm to achieve a perfect reconstruction with no errors. The same reasoning used before applies here as well.

Algorithm 1 AeTuner

Input: X_0 : Initial points, *n_iters* : The number of iterations
Output: *eval_xs* : The evaluated points in the tuning process

Phase 1 – Initialization

Train autoencoder for source domain
Train autoencoder for target domain

Phase 2 – Fill the Knowledge Base (KB)

Use the pretrained source model to encode in latent space the points in source domain and save them in the KB

Phase 3 – Tuning

Draw some points X_s from the KB and, if missing from the target number required, randomly generate some from the target domain and encode them
$URM \leftarrow$ CalculateURM(X_s)
eval_xs $\leftarrow \emptyset$ ▷ a list to contain all the already evaluated target points
for x in X_0 **do**
    Evaluate x, encode it with the target encoder and add the result to KB
    Add x to *eval_xs*
end for
while *lenght*(*eval_xs*) $<$ *n_iters* **do**
    X_s*_decoded* $\leftarrow$ X_s decoded in normalized target domain
    eval_xs_decoded $\leftarrow$ *eval_xs* normalized in the target domain
    weights $\leftarrow$ ComputeWeights(X_s*_decoded*, *eval_xs_decoded*)
    ranked_points $\leftarrow$ RankPoints(X_s*_decoded*, URM, *weights*)
    ranked_xs_decoded $\leftarrow$ *ranked_xs* decoded in target domain
    for *point* in *ranked_xs_decoded* **do**
        if *point* has been used before **then**
            Skip to next iteration
        else
            Evaluate *point*, encode it with the target encoder and add the result to KB
            Add *point* to *eval_xs*
        end if
    end for
end while
return *eval_xs*

Algorithm 2 AeNTTuner

Input: X_0 : Initial points, *n_iters* : The number of iterations
Output: *eval_xs* : The evaluated points in the tuning process

Phase 1 – Initialization

Train autoencoder for source domain
Initialize the target domain autoencoder weights using `xavier_uniform` with gain suitable for `LeakyReLU`

Phase 2 – Fill the Knowledge Base (KB)

Use the pretrained source model to encode in latent space the points in source domain and save them in the KB

Phase 3 – Tuning

Draw some points X_s from the KB and, if missing from the target number required, randomly generate some from the target domain and encode them
$URM \leftarrow$ CalculateURM(X_s)
eval_xs $\leftarrow \emptyset$ ▷ a list to contain all the already evaluated target points
for x in X_0 **do**
    Evaluate x, encode it with the target encoder and add the result to KB
    Add x to *eval_xs*
end for
while *lenght*(*eval_xs*) $<$ *n_iters* **do**
    if enough points in *eval_xs* for training and validation batch **then**
        Train the target model using early stopping as a regularization technique
    end if
    X_s*_decoded* $\leftarrow X_s$ decoded in normalized target domain
    eval_xs_decoded $\leftarrow$ *eval_xs* normalized in the target domain
    weights $\leftarrow$ ComputeWeights(X_s*_decoded*, *eval_xs_decoded*)
    ranked_points $\leftarrow$ RankPoints(X_s*_decoded*, URM, *weights*)
    ranked_xs_decoded $\leftarrow$ *ranked_xs* decoded in target domain
    for *point* in *ranked_xs_decoded* **do**
        if *point* has been used before **then**
            Skip to next iteration
        else
            Evaluate *point*, encode it with the target encoder and add the result to KB
            Add *point* to *eval_xs*
        end if
    end for
end while
return *eval_xs*

Chapter 4

Experimental Setup

This Chapter focuses on the presentation of the evaluation metrics we used to check on the effectiveness of our method in Section 4.1. Section 4.2 discussed the datasets we used for our experiments: how they were collected, how is their optimization space formed and what is the performance metric we aim to optimize.

All of our experiments and DNN trainings have been performed on a Lenovo ThinkPad laptop with Intel Core i5-10210U CPU and 16 GB of DDR4 memory.

4.1 Evaluation Metrics

To evaluate the results obtained by our method we considered IB and CR as our main evaluation metrics. We also consider the time taken for the execution of the single tuning iteration, but, since it is just a measure taken at runtime, we do not report it here.

4.1.1 Iterative Best (IB)

Iterative Best (IB) is a very simple metric: it shows the best result obtained by a configuration at a given iteration. Let us define $\boldsymbol{y_t}$ the vector of all the performance metrics obtained up to iteration t. It is customary to normalize the vector of results beforehand. In our experiments we normalize the metrics over the baseline configuration, *i.e.* the tunable parameters configuration we set to indicate the starting point for our optimization, and we then subtract one to the result. This procedure is equivalent to standard Normalized Performance Improvement (NPI). NPI is a measure for the achieved performance improvement over the maximum possible performance obtainable [6]. We perform this nor-

malization in order to have a comparable measure between different applications performance. We call the obtained vector $\tilde{\boldsymbol{y}}_{\boldsymbol{t}}$. With these conventions, the IB is simply defined as:

$$IB(t) = \max_{i=1...t} \tilde{\boldsymbol{y}}_{\boldsymbol{t}}$$

This simple metric is really useful to see the best performing configuration found in the training process up until a specific iteration. To evaluate how fast our model finds good configurations, we take advantage of the normalized sum of the IB vector over the number of total iterations N.

$$\hat{IB} = \frac{1}{N} \sum_{t} IB(t)$$

With this metric we can clearly see that a number very close to 1 indicates that the best possible configuration has been found early in the tuning process, moreover configurations with small improvements from the baseline have not been the best for many iterations.

4.1.2 Cumulative Reward (CR)

Cumulative Reward (CR) is an evaluation metric taken from the reinforcement learning domain. It is strongly linked to the idea of reward and regret which take part in the **online** feedback given to the RL agent after the interaction with the environment in order to reinforce (or punish) the last action or set of actions. It is formally defined as the cumulated sum of all the normalized results. Using the definition of normalized vector of results given in Section 4.1.1 we build the Cumulative Reward vector $\boldsymbol{CR}$ as:

$$CR_t = \sum_{i=1}^{t} \tilde{y}_i$$

Where CR_t is the t-th value in the CR vector and $\tilde{y}_i$ is the i-th value in the $\tilde{\boldsymbol{y}}$ vector. This metric is really useful to visualize how well the algorithm is performing mid-tuning since it shows how much the considered model has been able to leverage the tuning environment to get consistently good configurations. Similarly to Section 4.1.1, we use a normalized version of this metric at the last iteration to summarize the entirety of the tuning process. Let N be the total number of iterations performed by the tuning algorithm, $\hat{CR}_N$ is the CR value obtained at the last iteration divided by the total number of executed iterations

during tuning.

$$C\hat{R}_N = \frac{CR_N}{N}$$

Also in this case a value closer to 1 indicates that during the whole training process the model has made most of the time the perfect choice for the optimization. A value closer to -1 shows instead a poor set of choices since the algorithm visited mostly configurations that did not improve over the baseline.

Comparing the two metrics IB and CR, we use IB to indicate what is the best configuration obtained during the training process at a given iteration, whereas CR shows the behaviour of the algorithm during the entire training process keeping memory of all the choices (good or bad) made throughout the optimization procedure.

4.2 Datasets

In this work we used previously collected datasets for our experiments. In performance optimization we could directly apply our configurations to the real system running the application we want to optimize. This is clearly unfeasible since we want our experiments to be reproducible for a real system would be constantly affected by changes in workload and the simultaneous execution of other processes on the same machine. Moreover, in order to receive a feedback on the performance for our configurations would require a lot of time. Using a dataset removes this issues and makes the analysis of the algorithms much easier. Each dataset is composed by a set of benchmarks each one having a set of configurations, workloads and performance metrics relative to the configurations. Each dataset has a different regressor to simulate the application of the suggested configurations to the real system: cBench has a simple look up table for its configurations are enumerable, Cassandra and DaCapo need both a more complicated model that fits the limited amount of real data to predict reasonable values when prompted with unknown configurations.

4.2.1 cBench

The cBench suite [21] is a collection of open-source programs with multiple datasets assembled by the community to enable different workload executions and target different compilers as GCC and LLVM to name a few. In this suite the source code of every program is simplified in order to ease portability and

for this reason the suite has been used in many works on Iterative Compilation [6, 12]. In our work we use a dataset collected using this suite in [12]. Because this suite has been created to study the optimization of C language compilers its tunable parameters are flags that can be selected activated at compilation time. Because the only tunable knobs of a compiler are the optional flags, our tuning space is simply formed by binary flags which allow us to have an easily computable tuning domain size and lets us easily try all the possible flags combinations. The dataset we use has been created using GCC compiler, we hereby report the flags considered and what are their use according to the gcc documentation[1]:

- `-funsafe-math-optimization`: This mode enables optimizations that allow arbitrary reassociations and transformations with no accuracy guarantees. It also does not try to preserve the sign of zeros. This means that in floating point math expression like $x+(y+z)$ and $(x+y)+z$ are no longer equal.

- `-fno-guess-branch-probability`: Do not guess branch probabilities using heuristics. Generally GCC uses heuristics to predict branch probabilities based on the control flow graph. If not specified with the flag then they are decided by the optimization type selected `-O2` or `-O3`.

- `-fno-ivopts`: Suppresses high-level loop induction variable optimizations, which are enabled if `-O2` or `-O3` is used. These optimizations are generally profitable but, for some specific cases of loops with numerous uses of the iteration variable that follow a common pattern, they may end up destroying the regularity that could be exploited at a lower level and thus producing inferior code.

- `-fno-tree-loop-optimize`: Do not perform any optimization on loops trees.

- `-fno-inline-functions`: Suppresses automatic inlining of subprograms, which is enabled if `-O3` is used.

- `-funroll-all-loops`: Unroll all loops, even if their number of iterations is uncertain when the loop is entered. This usually makes programs run more slowly.

[1] https://gcc.gnu.org/wiki

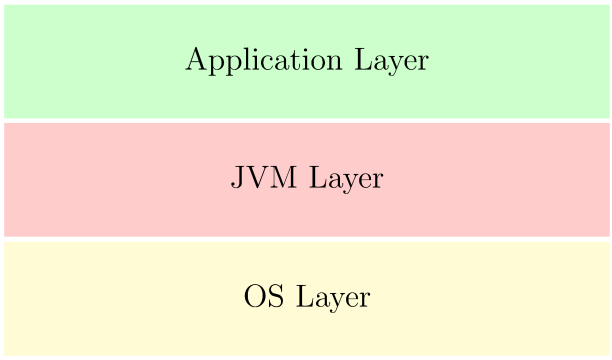


Figure 4.1: Application, JVM, and OS layers in Cassandra DBMS

- `-O2` or `-O3`: Using `-O2` GCC performs nearly all supported optimizations that do not involve a space-speed tradeoff. This flag increases compilation time and reduces execution time. `-O3` optimizes even more than `-O2` and in return it has generally lower execution times.

4.2.2 Cassandra

Apache Cassandra is a free and open-source, distributed, NoSQL database management system designed to handle large amounts of data across many commodity servers, providing high availability with no single point of failure. We choose this dataset because Cassandra is a widely used DBMS in many enterprise situations such as Netflix and Twitter. Moreover with this dataset we can work with parameters at different layers of the IT stack: we can modify parameters from the JVM, the OS and also some from the application layer. How the single layers are set on top of each other can be seen in Figure 4.1. The complete description on how this dataset was collected is the argument of Fabbrizio's thesis work [19]. In this work we limit ourselves at an high level description of the dataset's parameters. This dataset appears to have a more complex tunable domain than cBench with many parameters that can have great variability in the integer range and categorical values that make the tunable space much more difficult to be learnt and exploited. To have a good representation of the effect of the workload on the performance obtainable by a system running Cassandra, Yahoo! Cloud Servicing Benchmark (YCSB) has been used to simulate the effect of different workloads on the system. The dataset has been collected with an EC2 instance of AWS over the span of several months.

Application layer parameters

We hereby present the tunable parameters of the application layer in Cassandra. These are the most relevant parameters in the optimization process as noted in the Cassandra documentation.

- CompactionStrategy: Depending on the use case of each table in the database one can select the appropriate compaction strategy. The values are Size Tiered Compaction Strategy, good for insert-heavy and general workloads, Leveled Compaction Strategy for read-heavy workloads, Date Tiered Compaction Strategy, deprecated in Cassandra 3.08/3.7 and later because the Time Window Compaction Strategy is the improved version, ideal for immutable time series data.

- ConcurrentReads: Controls the maximum number of threads allocated to a particular stage. Having an optimal value will increase performance, but raising the value beyond the limit will decrease the DBMS performance. A good baseline value is 4 concurrent reads per processor core.

- ConcurrentWrites: Controls the maximum number of threads allocated for concurrent write operations. Writes are usually fast, for this reason it is not considered for tuning in most instances. A good value is generally above the one set for the reads.

- FileCacheSizeInMb: Represents the total memory to use for SSTable-reading buffers. The default value is set to 512 or smaller than $frac14$ of the Java Heap Memory.

- MemtableCleanupThreshold: Cassandra multiplies this threshold value with two other parameters, the Memtable heap space and the Memtable of heap space. The number obtained is the size of a Memtable to be flushed on the disk. Memtables are an in-memory data structure holding data before they are flushed to the disk, in a form of a Sorted Strings Table (SSTable), *i.e.* a file of key/value string pairs sorted by keys.

- CommitlogSync Period in ms: It is the time interval between syncing the commit log to disk. Without the commit log Cassandra would be really slow. When Memtables are written to disk we call them SSTables. SSTables are immutable, meaning once Cassandra writes them to disk it does not update them. When a column changes Cassandra needs to write a new

SSTable to disk, but doing it for every update would be very slow. To solve this issue Cassandra can write the updates in an in-memory queue, but if a machine goes offline all the data in the memory would be lost. Instead it writes updates on these commitlog, on the disk, which are optimized for writes. Default value is 10000.

- CommitLog Segment Size in MB: States the size of an individual commitlog file segment. A commitlog file may be archived, deleted or recycled after all its data has been flushed to SSTables. Default value is 32.

- CommitLog Compression: Commitlog can be compressed to reduce disk space usage. The default value is uncompressed. Possible values are LZ4Compressor, Snappy, Deflate.

- ConcurrentCompactors: The compaction procedure is used for different kinds of operations in Cassandra, the common thing about these operations is that they take one or more SSTables and output new SSTables. This procedure is needed due to the distributed nature of Cassandra. The number of concurrent compactors is the number of threads reserved by Cassandra to carry on these procedures. A small number of threads, compared to the amount of work, are going to slow down the entire database because they are not able to keep up with the work to do. Simultaneous compactions help preserve read performance in a mixed read-write workload by limiting the number of small SSTables that accumulate during a single long-running compaction. Increasing the number of concurrent compactors leads to more use of available disk space for compaction, because the procedure happens in parallel. The minimum is 2 and the maximum is 8 per core. The default value is computed as the fewest number of disks or cores.

- Compaction throughput MB per sec: Represents the MB per second to throttle compaction for the entire system. The faster the database inserts data, the faster the system must compact in order to keep the SSTable count down. The default value is 16 and a value of 0 will disable compaction throttling.

JVM layer parameters

The most important JVM parameters are reported here. These control the memory allocation and management of a Java application since many performance

bottlenecks for applications written in this language are due to inefficiencies in this area.

- MaxHeapSize: The maximum size of heap memory, allocated during the JVM startup. A small amount of memory needs a more frequent Garbage Collection cycle, which means that the application is stopped more frequently by the Garbage Collector, but every cycle requires less time compared to a higher size of memory.

- GcType: There are four different types of Garbage Collectors: Serial Garbage Collector, Parallel Garbage Collector, CMS Garbage Collector, G1 Garbage Collector. They are suitable for different case scenarios, based on the workload that the application is experiencing. The default GC is the Parallel Garbage Collector.

- NewRatio: Java uses generational garbage collection. This means that if you have an object, the more garbage collection events it survives, the further it gets promoted. It starts in the young generation (which itself is divided into multiple spaces - Eden, Survivor One and Survivor Two) and would eventually end up in the old generation if it survived long enough. NewRatio affects the internal proportion of heap memory, divided in young generation and old generation. The potential advantage is the fact that the young generation will grow and shrink automatically when the JVM dynamically adjusts the total heap size at run time. For example, with NewRatio = 3 the old generation will be three times as large as the young generation.

- ParallelGCThreads: Sets the number of threads used during parallel phases of the Garbage Collectors. The default value varies with the platform on which the JVM is running.

- ConcurrentGCThreads: Number of threads concurrent garbage collectors will use. The default value varies with the platform on which the JVM is running.

- SurvivorRatio: Sets a ratio for the size of internal memory parts of young and old generation.

- MaxTenuringThreshold: During young space garbage collection, every single object may be promoted in the young space or just moved to the old

space. For each object being copied, GC algorithm increases its age (number of collection survived) and if the age is above the current tenuring threshold it would be promoted to the old space. The actual tenuring threshold is dynamically adjusted by the JVM, but MaxTenuringThreshold sets an upper limit on it.

- CMSInitialOccupancyFraction: The JVM starts a GC cycle only when the heap is full, *i.e.* when there is not enough space available to store a newly allocated or promoted object. With the CMS Garbage Collector, it is not advisable to wait this long because the application keeps on running (and allocating objects) during concurrent GC. Thus, in order to finish a GC cycle before the application runs out of memory, the CMS Collector needs to start a GC cycle. For instance, CMSInitialOccupancyFraction = 75 means that the first CMS cycle starts when 75% of the old generation is occupied. Traditionally, the default value is 68, which was determined empirically during the initial releases of Cassandra.

OS layer parameters

The choice of OS parameters is more complex than the one performed for the application and JVM layers, simply because it is more difficult to individuate the most relevant parameters. Fabbrizio uses a random forest to individuate the most important OS parameters that are worth being optimized [19]. The final choice is reported in Table 4.1.

OS parameters
StorageReadAhead
NetworkNetIpv4TcpMaxSynBacklog
StorageQueueScheduler
MemoryVmDirtyExpire
MemoryTransparentHugepageEnabled
CPUSchedNrMigrate

Table 4.1: OS parameters considered in Cassandra dataset

4.2.3 DaCapo

The DaCapo suite [9, 10] is a collection of open-source benchmarks for in-depth testing of the Java Virtual Machine (JVM). This particular suite has the advan-

tage of allowing the allocation of large heap spaces, and for this reason it is really useful in studying the behaviour of the Garbage Collector (GC). This suite, as is the case with cassandra in Section 4.2.2, has the possibility to finely tune parameters for the JVM and OS, meaning that we can study separately the effect of different layers on the performance. The collection of this dataset is explained in Zenari's thesis [43] and has then been extended by Serafin [37]. Similarly as cassandra all the experiments have been performed on an EC2 instance of AWS. Also this dataset appears to be much more complex to be tuned when compared to cBench, many parameters in the tuning space can have categorical values or big ranges of integers.

JVM layer parameters

The parameters considered for the JVM are reported in Table 4.2, their explanation has already been given in Section 4.2.2 as they are exactly the same.

JVM parameters
MaxHeapSize
GcType
NewRatio
ParallelGCThreads
ConcurrentGCThreads
SurvivorRatio
MaxTenuringThreshold
CMSInitialOccupancyFraction

Table 4.2: The JVM parameters considered in the DaCapo dataset

OS layer parameters

The OS parameters have been selected with a random forest regressor in order to find the most useful in optimization by Zenari [43]. It comes to no surprise that the most relevant parameters are inherent to the CPU control in deciding the scheduling of the threads.

- MemoryTransparentHugepageEnabled: Enable/Disable huge pages.
- CPUSchedLatency: Time spent in scheduler tasks by the system.
- NetworkNetCoreSomaxconn: The maximum number of "backlogged sockets".

- MemoryVmDirtyExpire: How long something can be in cache before it needs to be written.
- MemoryVmDirtyWriteBack: How long before before data is flushed from memory to storage device.
- cpuSchedWakeupGranularity: The amount of time it must elapse before the scheduler can preempt the current task.
- CPUSchedMigrationCost: Amount of time after the last execution that a task is considered to be "cache hot" in migration decisions.
- NetworkNetIpv4TcpMaxSynBacklog: Max length of the SYN Backlog queue.
- MemoryVmDirtyBackgroundRatio: Decide how much modified data to keep in the cache.
- MemoryVmVfsCachePressure: Controls the tendency of the kernel to reclaim the memory which is used for caching of VFS caches.
- StorageReadAhead: The byte of data to pre-fetch.
- MemorySwappinnes Changes the balance between swapping out file cache pages in favour of anonymous page.
- StorageScheduler: Choose the storage scheduler.
- StorageRqAffinity: Control displacment of virtual disk.
- StorageQueueScheduler: Choose the queue scheduler.
- StorageNrRequests: Number of request scheduled for the storage.
- CPUSchedChildRunsFirst: Choose if the child process run before the parents.

Chapter 5

Results

In this chapter we are going to present the results obtained through a series of experiments designed to evaluate the performance of our proposed algorithms in different situations. In Section 5.1 we aim at evaluating the performance and the capability of transferring efficiently knowledge of our method when dealing with domains containing parameters controlling the same parts of the IT stack. In Section 5.2 we analyze the results of experiments with datasets containing more complex domains and cases where the source and target domain do not have a clear similarity. Section 5.3 tries to explain some still unanswered questions about the efficiency and the potential gains of using our method over others by focusing on experiments where the source and target domain are the same. Finally Section 5.4 briefly discusses the time our model needs to advance in the tuning process.

5.1 Trans Domain Experiments with Common Features Selection

The first experiment we conducted involves trying to transfer information between two datasets which have some parameters in common. We then considered transferring knowledge regarding the OS configuration between cassandra and dacapo. Owing to the fact that we focused only on a subset of all tunable parameters, we fixed the others to their baseline value, this choice led us to rescale the results in order to having the best possible configuration equal to the best possible parametrs for the OS and all the other values set to the baseline. The main question we try to answer with this experiment is whether or not the use of AE leads to an advantage in quality of the results and in tuning speed over a

simple random exploration in the target domain. As a secondary question, we try investigating if there is a more suitable AE model that gives a further advantage when using the novel method.

Scenario	Random	Ae with AE	AeNT with AE	Ae with VAE	AeNT with VAE
sunflow	0.67	0.48	**0.95**	0.36	0.22
lusearch-fix	0.84	0.70	**0.96**	0.58	0.36
avrora	**0.66**	0.53	0.52	0.56	0.23
h2	**0.00**	0.00	0.00	0.00	0.00
jython	0.82	0.91	**0.97**	0.74	0.24
xalan	0.71	0.67	**0.94**	0.11	0.66

Table 5.1: cassandra_dacapo_transfer - nanmedian over rep and nanmedian over env. Errors with nanmedian and nanstd. IB sum_norm

Scenario	Random	Ae with AE	AeNT with AE	Ae with VAE	AeNT with VAE
sunflow	0.16	0.15	**0.65**	−0.07	0.07
lusearch-fix	0.30	0.14	**0.83**	0.27	0.22
avrora	0.02	**0.23**	0.12	0.07	−0.06
h2	**0.00**	0.00	0.00	0.00	0.00
jython	0.50	0.70	**0.79**	0.69	0.11
xalan	−0.03	0.07	0.20	−0.04	**0.29**

Table 5.2: cassandra_dacapo_transfer - nanmedian over rep and nanmedian over env. Errors with nanmedian and nanstd. CR last_norm

Scenario	Ae	AeNT
sunflow		
lusearch-fix	**84**	
avrora	**27**	
h2		
jython	**4**	98
xalan		

(a) cassandra_dacapo_transfer CR best performing iteration last

Scenario	Ae	AeNT
sunflow		
lusearch-fix		
avrora	**84**	
h2		
jython	**9**	
xalan	96	**5**

(b) cassandra_dacapo_transferVAE CR best performing iteration last

Table 5.3: The first iteration after which Ae and AeNT tuners are able to consistently achieve an higher value for CR

The numerical results can be found in Tables 5.1-5.3. As can be seen, the proposed method is able to outperform the random search in almost all the considered benchmarks with the exception of "h2" which seems to be indifferent of the configuration tuning of its OS parameters. Another interesting finding is that the tuner with the target model that trains during the tuning process as previously described in 2 can achieve better performing configurations as opposed to

the pre-trained model. Looking jointly at Table 5.1 and Table 5.2, we can see that even though the fully pre-trained model has a normalized sum of IB lower than its competitors in many cases, it has the advantage of achieving a better CR at the last iteration and, more importantly, it can have a consistently better CR value than random from early iterations in Table 5.3 (as low as iteration 4 when looking at "jython"). This apparent incongruity may be explained by the update the AENT model receives multiple times during the training process, which improves the quality of the decisions made at the added cost of having to wait later iterations to get the benefit of those.

Speaking of the difference between the simple AE and the VAE, we noticed that the second one is not able to achieve as good results as its counterpart especially when looking at the quality of the configurations proposed. This may suggest that the type of model chosen for the DNN may be an important discriminant for the method's efficiency.

5.2 Trans Domain Experiments

Moving forward from the results achieved in Section 5.1, we tried to extend the tuning domain to all the tunable parameters of cassandra and dacapo datasets to discover if the autoencoders were able to find similarities between parameters that have apparently little to no similarities between each other. In fact we no longer limit our search to parameters controlling only the OS, but we simultaneously try to learn useful representation of parameters from the JVM.

Scenario	Random	Ae with AE	AeNT with AE	Ae with VAE	AeNT with VAE
sunflow	0.03	0.75	**0.97**	0.64	0.61
lusearch-fix	0.04	0.72	0.79	**0.99**	0.47
avrora	0.22	0.86	**0.94**	0.63	0.00
h2	0.22	0.96	0.89	0.70	**0.97**
jython	0.07	0.18	**0.93**	0.79	0.05
xalan	0.07	0.41	0.94	0.72	**0.98**

Table 5.4: cassandra_dacapo_transfer_nocommon - nanmedian over rep and nanmedian over env. Errors with nanmedian and nanstd. IB sum_norm

Looking at the results in Table 5.4 we can notice how previously untunable benchmarks such as "h2" can now be tuned with very good results, meaning that even though OS related parameters are not important for an improvement in the target metric, the method can automatically focus on changing the other ones, while a randomized search does not come nearly close to the perfect solution. By

Scenario	Random	Ae with AE	AeNT with AE	Ae with VAE	AeNT with VAE
sunflow	0.01	0.18	**0.19**	−0.01	−0.69
lusearch-fix	**0.02**	−0.56	−0.01	−0.03	−0.64
avrora	**−0.02**	−0.35	−0.27	−0.49	−0.71
h2	0.12	**0.52**	0.27	−0.03	−0.23
jython	0.04	−0.20	**0.07**	−0.35	−0.27
xalan	−0.01	−0.09	**0.15**	−0.53	−0.49

Table 5.5: cassandra_dacapo_transfer_nocommon - nanmedian over rep and nanmedian over env. Errors with nanmedian and nanstd. CR last_norm

Scenario	Ae with AE	AeNT with AE
Prova sunflow	11	**1**
Prova lusearch-fix		
Prova avrora		
Prova h2	**2**	**2**
Prova jython		**60**
Prova xalan		**1**

Table 5.6: cassandra_dacapo_transfer_nocommon CR best performing iteration last

a simple comparison between the results obtained by Random in Section 5.1 and the ones obtained in this more complex scenario, we can see that the optimization process has become more difficult for the simple random search, conversely our method improves on the previous results indicating that this scenario offers more possibilities to find similarities between the datasets. Another remark can be done comparing the results obtained using a VAE in this scenario and the one presented in 5.1: this time the model seems to catch good configurations early in the tuning process, sometimes well before the AE does. But if we look at the CR values at the last iteration reported in Table 5.5 we can see that the algorithm using the VAE gets very big negative values for this metric in all the considered benchmarks. This leads us to conclude that the algorithm based on the transfer performed by the VAE did not really learn how to suggest good configurations to try early on, but it may have found them later in the optimization process, thus it might have never had the chance to fully exploit them. It is also noticeable how the Auto Encoder with No pre-trained Target (AENT) algorithm can achieve in almost all the benchmarks better results than its fully pre-trained counterpart, suggesting that it may be a good idea to let the tuner *waste* some iterations to autonomously explore the space when dealing with datasets as complicated as these to get a better understanding of the structure of good configurations. Giving a strict comparison with the performance obtained by Random during tuning we get better CR values only when using the vanilla AE as the space encoding model. It is noticeable how the AENT algorithm can obtain higher

values for this metric even when completely untrained. This may show that the space transformation operated by the AE can give many benefits over just randomly selecting the configurations.

We also tried to transfer knowledge between two very structurally different domains, namely cBench as a source and cassandra as target, to discover if the proposed methodology could transfer any useful knowledge in a situation where the source and target domains are seemingly uncorrelated. In fact cBench is composed of only binary flags, conversely cassandra has many more parameters, many of which are real valued, have big integer intervals or are categorical. Since cassandra has 100 possible values for the workload (represented by the parameter `YCSB.runThreads`) we chose two of them far apart to see if the choice of workload would affect the quality of the results. As a matter of fact, we would expect a different workload, *i.e.* a different load on the system, to affect positively/negatively on the performance obtainable by a given configuration of tunable parameters. For this reason configurations that in one case lead to similar performances, if applied with the wrong workload, they could lead to an increase/decrease of the performance.

Scenario	Random	Ae with AE	AeNT with AE	Ae with VAE	AeNT with VAE
YCSB.runThreads= 10	0.82	**0.88**	0.83	0.60	0.71
YCSB.runThreads= 80	0.82	**0.88**	0.82	0.58	0.70

Table 5.7: cBench_cassandra_transfer - nanmedian over rep and nanmedian over env. Errors with nanmedian and nanstd. IB sum_norm

Scenario	Random	Ae with AE	AeNT with AE	Ae with VAE	AeNT with VAE
YCSB.runThreads= 10	0.29	0.21	**0.58**	0.18	0.25
YCSB.runThreads= 80	0.22	0.12	**0.55**	0.07	0.24

Table 5.8: cBench_cassandra_transfer - nanmedian over rep and nanmedian over env. Errors with nanmedian and nanstd. CR last_norm

Table 5.7 and Table 5.8 report the results of this experiment. It is clearly visible that also in this case VAE struggles to find good configurations in a timely manner even when compared to Random whereas the simple AE can suggest configurations with a better performance earlier than the randomized search. An important remark has to be done looking at the values of CR obtained at the last iteration where it is clearly visible that the AENT algorithm is making better quality suggestions than its competitions showing that it has achieved a better knowledge on how to leverage the parameter space.

5.3 Same Domain Experiments

In the previous sections we showed that the proposed algorithms improve on the quality of the tuning obtainable by exploring random configurations in a completely unknown space. There are still some questions that need to be answered:

1. Is the mapping on different common domain between source and target really beneficial when considering a tuning problem?

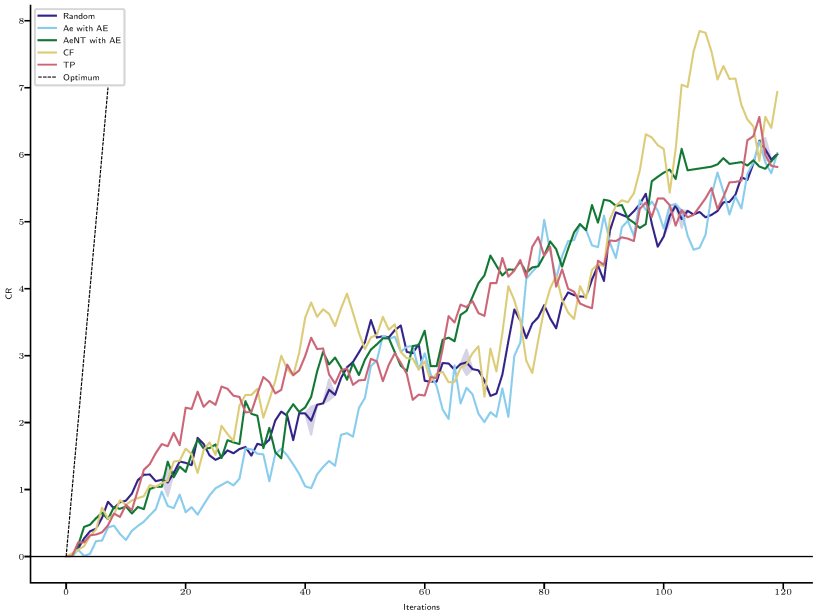
2. It is well known that autoencoders add some errors in the reconstruction process from the latent representation, how does this affect the tuning process?

3. Why is there such a big difference in the results when using a VAE instead of an AE?

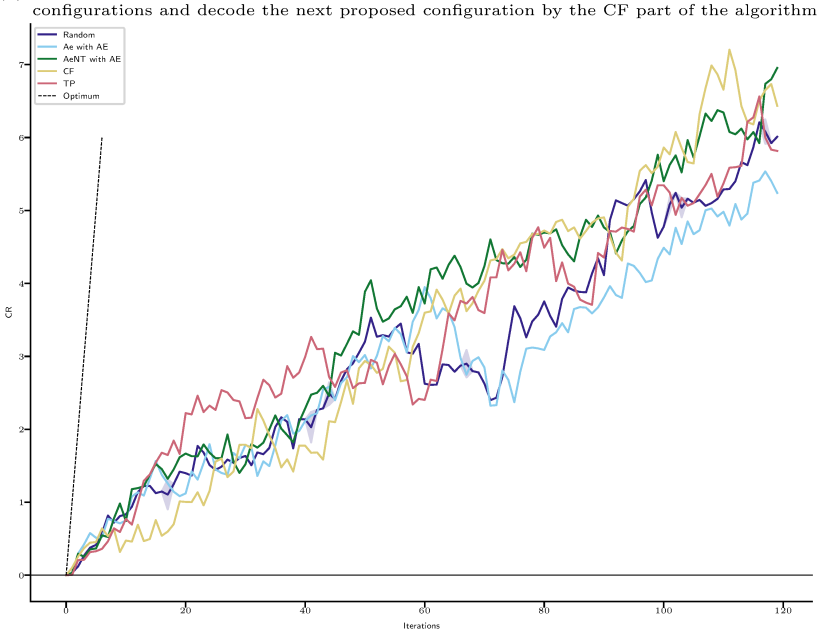
In order to answer these questions we set up different experiments aimed at tuning benchmarks[1] of the target dataset while giving as source all the other benchmarks from the same one. In this setup we were then able to compare the results obtained by the proposed algorithms with the ones of CF and TP since we do not have to worry about the differences in the source and target domains.

As a first experiment we applied all the previously mentioned tuning methods to cBench as it has a very simple domain structure consisting of only 7 binary flags which makes it possible and easy to count all the possible configurations that a tuner can explore during the tuning process. So as to answer question 2, we designed a slightly different version of our Algorithm 1 and 2 presented in Section 3.2 which keeps 2 parallel KBs, one for the points in the common encoded domain for source and target and one containing the points with the same indices, but in the original source domain in order to being able to make all the CF part of the algorithm in the common encoded domain (thus keeping the benefits, if any, of encoding the source and target space) and at the same time bypass the decoding portion of the autoencoder normally needed to obtain a valid parameter configuration. The graph of the CR over all the tuning iterations can be seen in Figure 5.1. These plots show some interesting findings: first of all by a simple comparison between 5.1a and 5.1b it is clearly visible how removing the decoding part to our algorithm immediately leads to higher values for the considered metric, this means that the decoding operation is surely responsible for the introduction

[1]Programs for which we have configuration, performance value pairs in a dataset



(a) CR values obtained averaging all benchmarks and workloads when using the AE to encode the configurations and decode the next proposed configuration by the CF part of the algorithm



(b) CR values obtained averaging all benchmarks and workloads when using the AE only for performing the encoding in the common space between source and target

Figure 5.1: Mean CR values across cBench dataset

of some errors that degrade the overall performance of the method. For this reason it is better to refer to Figure 5.1b as an *ideal optimal case* since this quality in the configurations reconstruction can not be achieved by an autoencoder. With this consideration in mind, we can focus our attention on Figure 5.1a to analyze if there is any advantage in using the proposed tuning algorithms. Although in the very first iterations there is no noticeable difference between the different methods, we can see that between iterations 60 and 90 AENT can achieve better

values for CR in many benchmarks, after this interval, vanilla CF is able to catch up. AE, on the other hand, does not perform in the same way and is in many points in the graph the worst performing method, even though in the very first iteration it seems to have a slight advantage. This reinforces the idea that after a rough start with a completely untrained autoencoder, AENT can effectively learn how to suggest better performing configurations by just exploring the target space and performing multiple learning sessions as opposed to a single monolithic training phase that necessitates the whole knowledge of the target space, which is most of the time completely unavailable.

Looking again at Figure 5.1b and comparing the CR values obtained with AENT and CF we can see that in the first part of the graph the encoding of the source and target space surely has a benefit in finding better configurations early on in the training process. This finding may suggest that the latent space created by the autoencoders in the innermost layer is better in highlighting the similarities in the source and target configurations even if the source and target domain are the same.

To assess how much the autoencoder distorts the reconstruction of the encoded samples we counted how many different configurations have been tried; since any benchmark in cBench has only 128 possible configurations, by performing the tuning in the same number of iterations we expect to try very single one of them. Condidering the vanilla AE, we see that the fully pre-trained model visits all the possible configurations everytime, whereas the model trained in multiple steps visits only 124 configurations averaging all the benchmarks and workloads. This suggests that in the tuning process the model gets stuck on fairly good configurations until it reaches the next training phase, where with new weights it is able to visit other configurations. Looking at the VAE the situation drastically changes: in fact the fully pre-trained model visits only a mean of 64 configurations, and the AENT model sees only a mean of 60 different configurations. Even with this strong lack of exploration, according to Table 5.9 the VAE obtains undoubtedly the best results when used in the AENT algorithm.

This result is interesting to us because in previous sections using the VAE led to poorer results. One possible explanation for this results improvement could be lying in the domain structure since in the cBench case there are only binary flags and in all the previous experiments there were also many parameters with real and a broader integer range of values. As a result of this particular domain structure, the hard cast to 0 or 1 performed after the decoding state may favor

Scenario	Random	Ae with AE	AeNT with AE	CF	TP	Ae with VAE	AeNT with VAE
automotive_bitcount	0.10 ± 0.11	0.10 ± 0.11	0.07 ± 0.11	0.11 ± 0.12	0.09 ± 0.12	-0.03 ± 0.21	$\mathbf{0.16 \pm 0.15}$
automotive_qsort1	0.18 ± 0.17	0.18 ± 0.16	0.15 ± 0.20	0.18 ± 0.17	0.16 ± 0.16	$\mathbf{0.26 \pm 0.24}$	0.12 ± 0.20
automotive_susan_c	0.23 ± 0.20	0.24 ± 0.20	0.17 ± 0.18	0.24 ± 0.20	0.24 ± 0.20	-0.06 ± 0.27	$\mathbf{0.34 \pm 0.26}$
automotive_susan_e	0.24 ± 0.20	0.21 ± 0.20	0.20 ± 0.19	0.24 ± 0.19	0.25 ± 0.19	0.08 ± 0.23	$\mathbf{0.26 \pm 0.25}$
automotive_susan_s	-0.05 ± 0.07	-0.05 ± 0.06	-0.08 ± 0.08	-0.05 ± 0.07	-0.06 ± 0.07	-0.17 ± 0.08	$\mathbf{-0.01 \pm 0.09}$
security_blowfish_d	0.01 ± 0.04	0.00 ± 0.04	$\mathbf{0.03 \pm 0.11}$	0.01 ± 0.04	0.02 ± 0.04	-0.11 ± 0.14	-0.13 ± 0.13
security_blowfish_e	-0.11 ± 0.46	-0.12 ± 0.46	-0.12 ± 0.39	$\mathbf{-0.10 \pm 0.46}$	-0.11 ± 0.46	-0.24 ± 0.43	-0.12 ± 0.53
security_rijndael_d	0.04 ± 0.16	0.05 ± 0.16	0.05 ± 0.14	0.04 ± 0.16	$\mathbf{0.06 \pm 0.16}$	0.00 ± 0.24	-0.07 ± 0.20
security_rijndael_e	-0.01 ± 0.21	-0.01 ± 0.21	-0.03 ± 0.16	0.01 ± 0.20	0.01 ± 0.20	0.02 ± 0.35	$\mathbf{0.05 \pm 0.20}$
security_sha	0.08 ± 0.06	0.07 ± 0.05	0.07 ± 0.08	0.07 ± 0.05	0.08 ± 0.05	0.03 ± 0.25	$\mathbf{0.15 \pm 0.12}$
telecom_adpcm_c	-0.08 ± 0.28	-0.08 ± 0.28	-0.11 ± 0.30	-0.08 ± 0.28	-0.07 ± 0.28	-0.13 ± 0.38	$\mathbf{-0.06 \pm 0.31}$
telecom_adpcm_d	-0.25 ± 0.23	-0.25 ± 0.23	-0.21 ± 0.23	-0.26 ± 0.22	-0.23 ± 0.23	-0.25 ± 0.30	$\mathbf{-0.16 \pm 0.25}$
telecom_CRC32	0.34 ± 0.06	0.33 ± 0.06	$\mathbf{0.38 \pm 0.07}$	0.34 ± 0.06	0.35 ± 0.06	0.06 ± 0.27	0.37 ± 0.08
consumer_jpeg_c	-0.06 ± 0.34	-0.06 ± 0.34	-0.18 ± 0.34	-0.07 ± 0.34	-0.06 ± 0.33	-0.29 ± 0.41	$\mathbf{-0.01 \pm 0.39}$
consumer_jpeg_d	$\mathbf{0.00 \pm 0.06}$	0.00 ± 0.07	0.00 ± 0.08	0.00 ± 0.07	0.00 ± 0.08	0.00 ± 0.09	0.00 ± 0.11
consumer_tiff2bw	-0.08 ± 0.20	-0.11 ± 0.20	-0.19 ± 0.22	-0.11 ± 0.20	-0.10 ± 0.19	-0.31 ± 0.37	$\mathbf{-0.07 \pm 0.24}$
consumer_tiff2rgba	-0.18 ± 0.16	-0.20 ± 0.15	-0.20 ± 0.16	$\mathbf{-0.17 \pm 0.16}$	-0.19 ± 0.16	-0.36 ± 0.20	-0.26 ± 0.15
consumer_tiffdither	0.13 ± 0.14	0.14 ± 0.15	0.01 ± 0.13	0.09 ± 0.13	0.13 ± 0.14	$\mathbf{0.37 \pm 0.28}$	0.05 ± 0.24
consumer_tiffmedian	0.00 ± 0.35	0.00 ± 0.35	0.00 ± 0.36	$\mathbf{0.00 \pm 0.34}$	0.00 ± 0.35	-0.01 ± 0.36	0.00 ± 0.38
network_dijkstra	0.32 ± 0.11	0.34 ± 0.10	0.30 ± 0.08	0.33 ± 0.10	0.33 ± 0.10	0.39 ± 0.17	$\mathbf{0.41 \pm 0.09}$
network_patricia	0.36 ± 0.09	0.36 ± 0.09	0.44 ± 0.10	0.37 ± 0.09	0.36 ± 0.09	$\mathbf{0.54 \pm 0.16}$	0.20 ± 0.15
office_stringsearch1	0.21 ± 0.36	0.20 ± 0.36	0.21 ± 0.38	0.17 ± 0.36	0.18 ± 0.35	0.25 ± 0.51	$\mathbf{0.35 \pm 0.44}$
bzip2d	0.06 ± 0.20	0.06 ± 0.19	0.05 ± 0.22	0.05 ± 0.20	0.05 ± 0.20	$\mathbf{0.20 \pm 0.27}$	0.04 ± 0.20
bzip2e	0.08 ± 0.29	0.10 ± 0.28	0.12 ± 0.28	0.10 ± 0.28	0.10 ± 0.28	$\mathbf{0.32 \pm 0.27}$	0.22 ± 0.28

Table 5.9: cBench - nanmedian over rep and nanmedian over env. Errors with nanmedian and nanstd. CR last_norm

the mean biased representation typical of the VAE model. As a matter of fact, the VAE supposes that the input data follows a normal distribution, which is not the case in our numerical domain. It is in fact well known that, especially in the case of images, the reconstructed samples are *"blurrier"* than the original ones since the aim of the VAE is to being able to generate coherent samples from the latent space even if they were not seen in the training dataset.

To further test this supposition we tried optimizing the Branin function [8] which is commonly used in the literature as a synthetic optimization problem. To highlight the differences in the results obtained with different domains, we tested the AE and VAE with both the traditional branin function and one having only evaluations at integer values for its parameters. As expected, the VAE was able to achieve better results than the AE only when dealing with a domain consisting of only integer numbers. Looking at the number of visited configurations it is also possible to see that, when dealing with the real-valued domain, the VAE visits far less configurations than when in a discrete domain. This suggests that the VAE may gain an advantage when dealing with domains containing only a small range of integer values thanks to the hard cast performed after the decoding portion of the algorithm which forces the exploration of new configurations especially when the decoded value is near a boundary between two values. Conversely, a more complex domain containing ample range of possible values does not allow for this hard-casting mechanism which inevitably leads to a lower quantity of generally lower quality configurations being visited in the tuning process.

5.4 Execution Times

Speaking of the execution times for a single iteration, they vary accordingly to the complexity of the autoencoder model and the complexity of the domain to be tuned. In all our experiments the time taken for a single iteration was well below the 1 second roof, which means that in production this time is almost negligible. The only iterations that lied around 2 to 3 seconds were the ones in which AENT trained its target model. Figure 5.2 shows the time taken by the considered algorithms during a typical training session. We expect to have Ae and AeNT tuners almost overlapping, this does not happen. The main issue may be due to thermal throttling of the CPU since all our calculations have been performed on a laptop which lacks efficient refrigeration systems. This phenomenon may explain why AeTuner, which was trained after AeNTTuner, has longer iteration times. Another reason may be found in the aleatoriety of the deep learning model that when fully trained has to perform slightly longer calculations. The main part of time consumption of our approach lies in the training process of the DNN models: in fact, depending on the complexity of the domains and the dimensionality of the datasets it may be necessary a long amount of time in order to obtain a fully trained autoencoder, especially when we decide to explore the parameters' space to obtain an architecture with better performances than a generic one. If we plan to reuse a previously trained model this added cost is easily absorbed over an extended set of performed tests, otherwise it may be unfeasible to invest in this long process to perform a single optimization for in the same amount of time Random will inevitably explore a large part of the tunable domain.

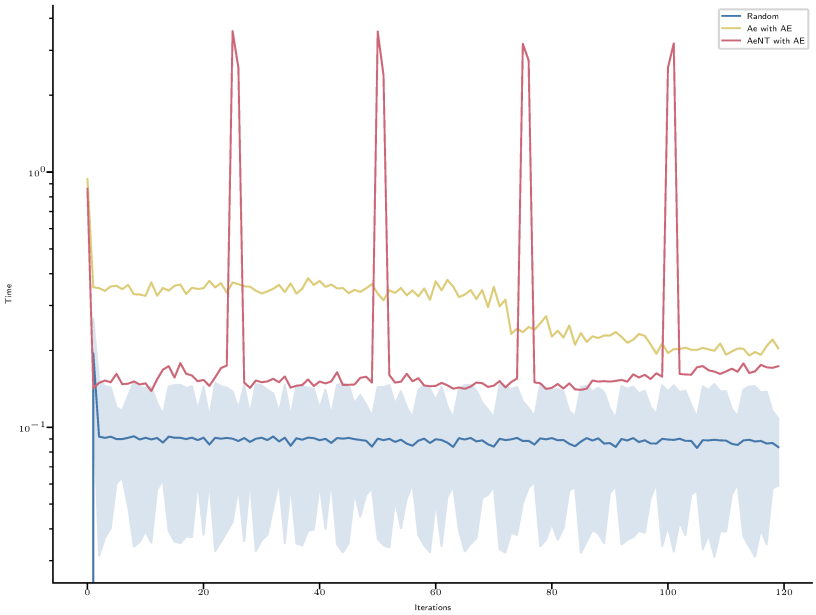


Figure 5.2: Execution times for each iteration of a typical execution. Spikes for AeNT algorithm have to be expected since at those iterations the target model is performing a training epoch

Chapter 6

Conclusions

In this thesis work we have presented a new procedure to transfer knowledge between tuning domains using DNN models, namely AEs and VAEs. To test our results we opted for using already collected datasets that contained data from famous open-source benchmark suites (cBench, DaCapo) and a widely used DBMS system (Cassandra). The datasets we have used allowed us to test our method in different conditions with varying types of domains and performance metrics to optimize. Our method effectively leverages Collaborative Filtering concepts to suggest new relevant tuning configurations from a Knowledge Base that contains points from the source domain and already evaluated points in the target domain mapped into a common domain space created by the Auto Encoders innermost latent layer. Even though our algorithm still has to face a reconstruction error due to the stochastic imperfections of the DNN models, we are still able to obtain good results when comparing our results with the ones obtained with state of the art methods in Section 5.3. Our method is shown to outperform a Random search in the target space for transfer learning applications when using a deep denoising autoencoder. This means that our approach learns the similarities between the source and target domains effectively and thus improves the speed of the tuning procedure. We also show that better results can be achieved if we train our DNN model during the tuning process instead of having a monolithic training session beforehand and thus always working with a fully pretrained target model, which is also very difficult to be done in a real world scenario when we rarely have access to many data points for the target domain. We have also discussed why the VAE does not have any considerable improvement over the simpler AE in Section 5.3. We also analyzed the time taken by our method to produce a valid configuration to be tested in the system to be tuned. We found that with a

sub-second time this procedure can happen almost instantaneously in a real life scenario, thus becoming negligible in comparison to the actual performance test (which approximately lasts for 60 to 90 minutes).

In a future extension of this work more DNN models should be analyzed, moreover different training procedures, for instance GAN-like methods may benefit the reconstruction quality from the latent space representation. An effert should also be made in order to apply a different method to select new samples in the common space: we think that a good alternative to Collaborative Filtering could be an approach based on Contextual BO or some approach leveraging information theoretic concepts as Age of Information.

Bibliography

[1] F. Agakov et al. “Using machine learning to focus iterative optimization”. In: *International Symposium on Code Generation and Optimization (CGO’06)*. 2006, 11 pp.–305. DOI: 10.1109/CGO.2006.37.

[2] Takuya Akiba et al. “Optuna: A Next-generation Hyperparameter Optimization Framework”. In: *Proceedings of the 25rd ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*. 2019.

[3] Rafeef M Al Baity, Esraa H Alwan, and Ahmed BM Fanfakh. “A Content Based Filtering Approach for the Automatic Tuning of Compiler Optimizations”. In: *Turkish Journal of Computer and Mathematics Education (TURCOMAT)* 12.6 (2021), pp. 3913–3922.

[4] Jason Ansel et al. “OpenTuner: An Extensible Framework for Program Autotuning”. In: *International Conference on Parallel Architectures and Compilation Techniques*. Edmonton, Canada, Aug. 2014. URL: http://groups.csail.mit.edu/commit/papers/2014/ansel-pact14-opentuner.pdf.

[5] Saba Anwar et al. “Using Association Rules to Identify Similarities between Software Datasets”. In: *2012 Eighth International Conference on the Quality of Information and Communications Technology*. 2012, pp. 114–119. DOI: 10.1109/QUATIC.2012.66.

[6] Amir Hossein Ashouri et al. “COBAYN: Compiler Autotuning Framework Using Bayesian Networks”. In: *ACM Trans. Archit. Code Optim.* 13.2 (2016). ISSN: 1544-3566. DOI: 10.1145/2928270. URL: https://doi.org/10.1145/2928270.

[7] Rémi Bardenet et al. “Collaborative hyperparameter tuning”. In: *Proceedings of the 30th International Conference on Machine Learning*. Ed. by Sanjoy Dasgupta and David McAllester. Vol. 28. Proceedings of Machine Learning Research 2. Atlanta, Georgia, USA: PMLR, June 2013, pp. 199–207. URL: https://proceedings.mlr.press/v28/bardenet13.html.

[8] Derek Bingham. *Branin Function*. URL: https://www.sfu.ca/~ssurjano/branin.html.

[9] S. M. Blackburn et al. "The DaCapo Benchmarks: Java Benchmarking Development and Analysis". In: *OOPSLA '06: Proceedings of the 21st annual ACM SIGPLAN conference on Object-Oriented Programing, Systems, Languages, and Applications*. Portland, OR, USA: ACM Press, Oct. 2006, pp. 169–190. DOI: http://doi.acm.org/10.1145/1167473.1167488.

[10] S. M. Blackburn et al. *The DaCapo Benchmarks: Java Benchmarking Development and Analysis (Extended Version)*. Tech. rep. TR-CS-06-01. http://www.dacapobench.org. ANU, 2006.

[11] John Cavazos et al. "Rapidly Selecting Good Compiler Optimizations using Performance Counters". In: *International Symposium on Code Generation and Optimization (CGO'07)*. 2007, pp. 185–197. DOI: 10.1109/CGO.2007.32.

[12] Stefano Cereda et al. "A Collaborative Filtering Approach for the Automatic Tuning of Compiler Optimisations". In: *The 21st ACM SIGPLAN/SIGBED Conference on Languages, Compilers, and Tools for Embedded Systems*. LCTES '20. London, United Kingdom: Association for Computing Machinery, 2020, pp. 15–25. ISBN: 9781450370943. DOI: 10.1145/3372799.3394361. URL: https://doi.org/10.1145/3372799.3394361.

[13] Stefano Cereda et al. "CGPTuner: A Contextual Gaussian Process Bandit Approach for the Automatic Tuning of IT Configurations under Varying Workload Conditions". In: *Proc. VLDB Endow.* 14.8 (2021), pp. 1401–1413. ISSN: 2150-8097. DOI: 10.14778/3457390.3457404. URL: https://doi.org/10.14778/3457390.3457404.

[14] Olivier Chapelle and Lihong Li. "An Empirical Evaluation of Thompson Sampling". In: *Advances in Neural Information Processing Systems*. Ed. by J. Shawe-Taylor et al. Vol. 24. Curran Associates, Inc., 2011. URL: https://proceedings.neurips.cc/paper/2011/file/e53a0a2978c28872a4505bdb51db06dc-Paper.pdf.

[15] Yang Chen et al. "Deconstructing Iterative Optimization". In: *ACM Trans. Archit. Code Optim.* 9.3 (Oct. 2012). ISSN: 1544-3566. DOI: 10.1145/2355585.2355594. URL: https://doi.org/10.1145/2355585.2355594.

[16] Eleftherios-Iordanis Christoforidis, Sotirios Xydis, and Dimitrios Soudris. "CF-TUNE: Collaborative Filtering Auto-Tuning for Energy Efficient Many-Core Processors". In: *IEEE Computer Architecture Letters* 17.1 (2018), pp. 25–28. DOI: 10.1109/LCA.2017.2716919.

[17] Paolo Cremonesi, Yehuda Koren, and Roberto Turrin. "Performance of Recommender Algorithms on Top-n Recommendation Tasks". In: *Proceedings of the Fourth ACM Conference on Recommender Systems*. RecSys '10. Barcelona, Spain: Association for Computing Machinery, 2010, pp. 39–46. ISBN: 9781605589060. DOI: 10.1145/1864708.1864721. URL: https://doi.org/10.1145/1864708.1864721.

[18] Lixin Duan, Dong Xu, and Ivor Tsang. *Learning with Augmented Features for Heterogeneous Domain Adaptation*. 2012. DOI: 10.48550/ARXIV.1206.4660. URL: https://arxiv.org/abs/1206.4660.

[19] Francesco Fabbrizio. "PerformanceHawk: Reinforcement Learning Applied to Automatic Online Performance Optimization". Sept. 2019.

[20] Matthias Feurer, Jost Springenberg, and Frank Hutter. "Initializing Bayesian Hyperparameter Optimization via Meta-Learning". In: *Proceedings of the AAAI Conference on Artificial Intelligence* 29.1 (Feb. 2015). DOI: 10.1609/aaai.v29i1.9354. URL: https://ojs.aaai.org/index.php/AAAI/article/view/9354.

[21] Grigori Fursin and Olivier Temam. "Collective Optimization: A Practical Collaborative Approach". In: *ACM Trans. Archit. Code Optim.* 7.4 (Dec. 2011). ISSN: 1544-3566. DOI: 10.1145/1880043.1880047. URL: https://doi.org/10.1145/1880043.1880047.

[22] Xavier Glorot and Yoshua Bengio. "Understanding the difficulty of training deep feedforward neural networks". In: *Proceedings of the Thirteenth International Conference on Artificial Intelligence and Statistics*. Ed. by Yee Whye Teh and Mike Titterington. Vol. 9. Proceedings of Machine Learning Research. Chia Laguna Resort, Sardinia, Italy: PMLR, May 2010, pp. 249–256. URL: https://proceedings.mlr.press/v9/glorot10a.html.

[23] Ian Goodfellow, Yoshua Bengio, and Aaron Courville. *Deep Learning*. http://www.deeplearningbook.org. MIT Press, 2016.

[24] Irina Higgins et al. "beta-VAE: Learning Basic Visual Concepts with a Constrained Variational Framework". In: *International Conference on Learning Representations*. 2017. URL: https://openreview.net/forum?id=Sy2fzU9gl.

[25] Daniel Jiwoong Im et al. *Denoising Criterion for Variational Auto-Encoding Framework*. 2015. DOI: 10.48550/ARXIV.1511.06406. URL: https://arxiv.org/abs/1511.06406.

[26] M. Kac and A. J. F. Siegert. "An Explicit Representation of a Stationary Gaussian Process". In: *The Annals of Mathematical Statistics* 18.3 (1947), pp. 438–442. DOI: 10.1214/aoms/1177730391. URL: https://doi.org/10.1214/aoms/1177730391.

[27] Michael Kampffmeyer et al. "The deep kernelized autoencoder". In: *Applied Soft Computing* 71 (2018), pp. 816–825. ISSN: 1568-4946. DOI: https://doi.org/10.1016/j.asoc.2018.07.029. URL: https://www.sciencedirect.com/science/article/pii/S1568494618304174.

[28] Alireza Karbalayghareh, Xiaoning Qian, and Edward R. Dougherty. "Optimal Bayesian Transfer Learning". In: *IEEE Transactions on Signal Processing* 66.14 (2018), pp. 3724–3739. DOI: 10.1109/TSP.2018.2839583.

[29] Diederik P Kingma and Max Welling. *Auto-Encoding Variational Bayes*. 2013. DOI: 10.48550/ARXIV.1312.6114. URL: https://arxiv.org/abs/1312.6114.

[30] Diederik P. Kingma and Max Welling. "An Introduction to Variational Autoencoders". In: *Foundations and Trends® in Machine Learning* 12.4 (2019), pp. 307–392. DOI: 10.1561/2200000056. URL: https://doi.org/10.1561%2F2200000056.

[31] Andreas Krause and Cheng Ong. "Contextual Gaussian Process Bandit Optimization". In: *Advances in Neural Information Processing Systems*. Ed. by J. Shawe-Taylor et al. Vol. 24. Curran Associates, Inc., 2011. URL: https://proceedings.neurips.cc/paper/2011/file/f3f1b7fc5a8779a9e618e1f23a7b7860-Paper.pdf.

[32] S. Kullback and R. A. Leibler. "On Information and Sufficiency". In: *The Annals of Mathematical Statistics* 22.1 (1951), pp. 79–86. DOI: 10.1214/aoms/1177729694. URL: https://doi.org/10.1214/aoms/1177729694.

[33] Wen Li et al. “Learning With Augmented Features for Supervised and Semi-Supervised Heterogeneous Domain Adaptation”. In: *IEEE Transactions on Pattern Analysis and Machine Intelligence* 36.6 (2014), pp. 1134–1148. DOI: 10.1109/TPAMI.2013.167.

[34] Sinno Jialin Pan and Qiang Yang. “A Survey on Transfer Learning”. In: *IEEE Transactions on Knowledge and Data Engineering* 22.10 (2010), pp. 1345–1359. DOI: 10.1109/TKDE.2009.191.

[35] Paul Resnick and Hal R. Varian. “Recommender Systems”. In: *Commun. ACM* 40.3 (Mar. 1997), pp. 56–58. ISSN: 0001-0782. DOI: 10.1145/245108.245121. URL: https://doi.org/10.1145/245108.245121.

[36] Francesco Ricci, Lior Rokach, and Bracha Shapira. “Recommender Systems: Techniques, Applications, and Challenges”. In: *Recommender Systems Handbook*. Ed. by Francesco Ricci, Lior Rokach, and Bracha Shapira. New York, NY: Springer US, 2022, pp. 1–35. ISBN: 978-1-0716-2197-4. DOI: 10.1007/978-1-0716-2197-4_1. URL: https://doi.org/10.1007/978-1-0716-2197-4_1.

[37] Mario Serafin. “Java Performance Variations: a Study on the Impact of Just-In-Time Compilation and Garbage Collection on Benchmarks Repeatability”. July 2021.

[38] Bobak Shahriari et al. “Taking the Human Out of the Loop: A Review of Bayesian Optimization”. In: *Proceedings of the IEEE* 104.1 (2016), pp. 148–175. DOI: 10.1109/JPROC.2015.2494218.

[39] Xiaoxiao Shi et al. “Transfer Learning on Heterogenous Feature Spaces via Spectral Transformation”. In: *2010 IEEE International Conference on Data Mining*. 2010, pp. 1049–1054. DOI: 10.1109/ICDM.2010.65.

[40] Tinu Theckel Joy et al. “A flexible transfer learning framework for Bayesian optimization with convergence guarantee”. In: *Expert Systems with Applications* 115 (2019), pp. 656–672. ISSN: 0957-4174. DOI: https://doi.org/10.1016/j.eswa.2018.08.023. URL: https://www.sciencedirect.com/science/article/pii/S0957417418305311.

[41] Pengfei Wei et al. “Easy-but-effective Domain Sub-similarity Learning for Transfer Regression”. In: *IEEE Transactions on Knowledge and Data Engineering* (2020), pp. 1–1. DOI: 10.1109/TKDE.2020.3039806.

[42] Karl Weiss, Taghi Khoshgoftaar, and DingDing Wang. "A survey of transfer learning". In: *Journal of Big Data* 3 (May 2016). DOI: 10.1186/s40537-016-0043-6.

[43] Giovanni Zenari. "Automatic performance tuning with past knowledge". Dec. 2019. URL: http://hdl.handle.net/10589/152263.

[44] Zhuangdi Zhu, Kaixiang Lin, and Jiayu Zhou. "Transfer Learning in Deep Reinforcement Learning: A Survey". In: *CoRR* abs/2009.07888 (2020). arXiv: 2009.07888. URL: https://arxiv.org/abs/2009.07888.